

A NEW DATA STRUCTURE
FOR THE
MULTI-USER TEXT EDITORS

A Thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in
Computer Science
in the
University of Canterbury
by
Mostafa Rajabian

University of Canterbury
1983

ACKNOWLEDGEMENTS

I would like to express my deepest gratitudes to my supervisor, Dr. M. A. Maclean, for his continuing guidance and very valuable suggestions and comments throughout this project.

I would also like to thank other staff of Computer Science Department who were helpful during the project. In particular, I thank Dr. R. E. M. Cooper who initiated this project and encouraged me to do it, and Dr. B. J. McKenzie for his useful suggestions during the period he was my supervisor. My thanks are also due to Professor J. E. L. Peck for his fruitful discussions about this project.

I am grateful to my family, especially my mother, who made it possible for me to come to New Zealand and encouraged me during the project by letters and telephones.

My final thanks are to all my friends in New Zealand whose friendship I have enjoyed during my studies here.

CONTENTS

CHAPTER	PAGE
ABSTRACT	1
1. INTRODUCTION	3
2. THE HISTORICAL DEVELOPMENT OF EDITOR DATA STRUCTURES	8
2.1 Batch editing systems	9
2.2 Sequential editing systems	11
2.3 Direct editing systems	13
2.4 Data structures using linked list of pages	19
2.5 Data structures using random access pages	25
2.6 Data structures using an array of line pointers	28
3. THE CHEF EDITOR DATA STRUCTURE	31
3.1 General description of the data structure	31
3.2 The implementation of the main commands	35
3.2.1 Accessing a line	35
3.2.2 Insertion	37
3.2.3 Deletion	39
3.3 The implementation of other important commands	41
3.4 The interface with the underlying file system	45
4. A NEW EDITOR DATA STRUCTURE	50
4.1 General characteristics	50
4.2 General structure	52
4.3 Detailed description of the data structure	55
4.4 The implementation of the main commands	60
4.4.1 Accessing a line	60
4.4.2 Insertion	62
4.4.3 Deletion	66

5. EXTENDED FEATURES OF THE NEW DATA STRUCTURE	69
5.1 Backup for the Undo command	69
5.2 Stacking of the unused blocks	72
5.3 Garbage collection	74
5.4 Recovery from system failure	77
6. IMPLEMENTATION OF THE NEW DATA STRUCTURE AND THE RESULTS	80
6.1 Implementation of the new data structure	80
6.2 Experimental results and comparisons	87
7. CONCLUSION AND THE FUTURE	104
7.1 CONCLUSION	104
7.2 THE FUTURE	106
REFERENCES	108
APPENDIX	110

LIST OF FIGURES

FIGURE		PAGE
2.1	Changes on the map after inserting an string	17
2.2	Structure of a page in ATS Editor	20
2.3	Movement of lines in ATS Editor	22
2.4	System lists after initialization of storage	24
2.5	Structure of a file list during editing	25
2.6	Workfile structure in GODOT Editor	29
3.1	Conceptual view of the CHEF data structure	32
3.2	Creating a gap for an insertion in CHEF	39
3.3	Layout of the pointer place in CHEF	44
3.4	Structure of a DAM file on the PRIME 750	46
3.5	Structure of a Random file on the ECLIPSE	47
4.1	Conceptual view of the new data structure	53
4.2	Pointer blocks Directory	60
4.3	Current window with its current information	61
4.4a	Changes in current and old windows for insertion	65
4.4b	Directory and Pointer blocks before and after insertion	65
4.5	Directory and Pointer blocks before and after deletion	67
5.1	Structure of Backup blocks for undo command	70

LIST OF TABLES

TABLE		PAGE
I	Number of new entries created for insert and delete	18
II	Maximum Workfile sizes for different line spacings	34
III	Number of i/o calls for an access in CHEF	37
IV	A pointer bit allocations and its effects in file size and map_table size	58
V	Results for typical access commands in a 1000-line file	90
VI	Results for typical access commands in a 3000-line file	90
VII	Results for inserting an external file in a 1000 and 3000-line file	92
VIII	Results for inserting from another region of Workfile for a 1000-line file	93
IX	Results for inserting from another region of Workfile for a 3000-line file	94
X	Results for delete commands in a 1000-line file	95
XI	Results for delete commands in a 3000-line file with detailed number of i/o	95
XII	Results for undo commands in a 1000-line file	98
XIII	Results for undo command in a 3000-line file	99
XIV	Effect of Garbage Collection in pointer values	100
XV	Overheads of Garbage Collection in a 1000-line file	102
XVI	Overheads of Garbage Collection in a 3000-line file	103

LIST OF GRAPHS

GRAPH		PAGE
I	Deleting a single line at different locations of a 3000-line file	96
II	inserting the last line in the middle of different size files	97

ABSTRACT

This thesis describes the design and implementation of a new and efficient data structure used for the workfiles of multiuser line-oriented text editors. One of the design objectives is the ability to use the editor (i.e. CHEF) as an editing server in a local ring network with a dedicated disk for the storage of users' workfiles. This objective is facilitated because the data structure allows random disk blocks to be used for the workfiles. This allows the editor to exist stand-alone without the need for an underlying file system.

The structure of the workfiles is a determining factor for the efficiency of storage and speed of any editor and it is just as important as the command language and other user level features of editors. (historical development of such data structures in a number of editors is described, and 6 general categories are developed. The advantages and disadvantages of each category are discussed and the main problems in the design of editor data structures are distinguished. The CHEF editor data structure, which solves most of these problems, is studied in detail. The new data structure, which is in fact a logical development of the CHEF data structure, brings further improvements and increases the performance. This is shown by experimental results.

Some advanced features which are facilitated by the new data structure are: backup of pointers for undoing a command, garbage collection of text storage space and unused

pointer values, and a recovery technique.

The data structure can be used in two different environments: on top of a file system, and with a dedicated disk. However it will show a better performance with a dedicated disk.

CHAPTER 1

INTRODUCTION

The original stimulus for this thesis was the idea of having an 'editing server' in one node of the local ring network being developed at the Department of Computer Science.

The editing server would consist of a small computer (probably a microprocessor), which would be responsible for servicing the editing requests of all users from any node of the network. At the beginning of the editing session, after the user requested to edit a file, the editing server would get a copy of the original file from the 'file server', which would be responsible for the storage and maintenance of all network users and systems files, and store it in its own 'dedicated disk' as the user's 'work file'. At the end of the editing session, with the user's request, this edited work file would be sent back to the file server, replacing the original file.

The idea of the editing server was first suggested by the designers of the Cambridge Ring Network, following their successful development of a file server for the network. But, to the knowledge of the author, there has been no attempt to develop such ^{an editing} ~~a~~ server.

The basic editor selected as the server for the Department of Computer Science ring network is 'CHEF' (Peck and Maclean 1981). It was realized from the beginning that

a most important consideration in the adaptation of CHEF (or any other editor) as the editing server is the data structure used for the users' work files.

It is highly desirable for this data structure to have the following characteristics:

1. It should provide a faster response time to the editing commands, compared to the existing data structures, under similar environments.
2. It should have the capability of utilizing its own disk space (a dedicated disk), without any requirement for an underlying file system.
3. In a multiuser environment, with the absence of a sophisticated operating system, the data structure should be:

--Easy to implement

--Feasible with a limited main memory.

In this thesis, we will concentrate on the editor data structures, and will present the design and implementation of a new data structure for this application.

With the development of on-line computing, and the advent of inexpensive terminals, the idea of on-line creation and modification of programs and texts has become widely accepted and used in the computer industry. Interactive text editors have now become an essential component of any small or large computing system.

An 'interactive text editor' is a computer program that is used for the creation and modification of texts. Most of the designers of text editors have considered the 'user level' of the editor to be of their most important aspect.

So the convenience of the user, and the power of the command language have become the most important design objectives. The implementation level objectives and techniques have been either ignored, or given less importance by the designers. As a result, there are numerous 'functional descriptions' of text editors in the literature, and little has been said about the implementation level. This has led to the development of advanced techniques and facilities at the user level. These include powerful command language facilities such as the use of macros, jumps, undo facilities, etc. There have even been some attempts to develop a design methodology for the command language. In this respect the language is divided into three parts: semantic, syntactic, and lexical components.

On the other hand only a few editor designers have realized the importance of the implementation level in the design of the editor. The most important element at this level is the structure of the text to be edited, or the 'work file', which is the manner in which the text is stored, accessed, and manipulated. Unlike an ordinary 'permanent' file, an editor work file is dynamically changing in an unpredictable way. Any portion of the file may be accessed at any time for editing. It grows quickly in size and complexity, expanding and contracting dynamically along its entire length. The data structure of an editor work file is important, since it affects the performance of the editor in the following ways:

1. The speed of the editor
2. The maximum allowable size of the text to be edited.
3. The number of the disk input/output operations.

4. Main memory usage.
5. The facilities provided at the user level.

In the study of a number of editor data structures, it was realized that most of them suffer from the problems that stem from ignoring one or more of the above factors in the design and implementation of the data structure. For example, the delete and insert commands may be very slow due to the extensive movement of the text within the work file, or the limit on the size of the file to be edited may be unacceptably small for some users.

The CHEF Editor, described in chapter 3, uses a data structure which solves most of these problems. The basic access, delete, and insert commands are implemented efficiently in this structure, so the general speed of the editor is fast. The maximum file size is only limited by the number of bits in a line pointer, rather than by memory capacity. The number of disk i/o operations is relatively small for most commands. The main memory usage with this structure is very economical, i.e. only 4 blocks of the file are resident at any time. At the user level, there are powerful and flexible commands and the data structure facilitates the implementation of some features such as the Undo command, and macro facilities.

The new data structure presented in this thesis is a logical improvement of the CHEF data structure. Some of the basic concepts of CHEF are preserved, while there are radical changes in others. The basic commands, specially deletion and insertion of text, are implemented differently, and more efficiently. The number of disk i/o operations is

effectively decreased. The main memory usage is slightly higher (7 blocks). The inherent structure of the new design also facilitates the implementation of backup for the Undo command, and increases its efficiency. It also uses a garbage collection algorithm which returns two garbage areas to the system: the unused text area, and more important, the unused logical block numbers. The latter gives the user the possibility of spending much longer time in a single editing session.

The layout of the chapters in this thesis is as follows: Chapter 2 describes the historical development of the data structures used in editors. It discusses the advantages and disadvantages of the data structures of some well known editors. Chapter 3 covers a detailed description of the CHEF data structure. In chapter 4, we will present the design of the new data structure. Chapter 5 gives the description of some extended features of the new data structure. This includes a new method of backup for Undo, stacking of unused blocks, garbage collection, and a recovery technique. Chapter 6 describes the implementation of the new data structure on CHEF, and gives some experimental results and comparisons. Chapter 7 is the conclusion of this thesis.

CHAPTER 2

THE HISTORICAL DEVELOPMENT OF EDITOR DATA STRUCTURES

In this chapter the historical development of editor data structures is studied. As mentioned earlier, while there are numerous behavioural descriptions of editors in the literature, there are only a few describing editor data structures. An attempt is made to study the data structure of some well known editors. General categories are presented for the historical development of these data structures, and it is believed that the data structures not discussed here fall into one of these categories. The data structures are discussed with respect to the following factors, as mentioned before:

1. The speed of the editor, which is directly dependent on the speed of the three basic commands of an editor: access, delete, and insert.
2. The maximum allowable size of the text to be edited. This is another important factor directly dependent on the data structure used.
3. The number of the disk input/output transfers. This is an important factor for editors which utilize the disk for the storage of the work file.
4. Main memory usage. The amount of main memory used for the work file buffers. This factor is more important for 'multiuser' editors, and also for editors implemented in small systems.

5. The data structure is also important, to some extent, in imposing restrictions or allowing extensions to the command language, and the facilities provided at the user level.

The three commands access, delete, and insert, mentioned in '1' above, are considered the most important commands of an editor for implementation purposes. The reason for this selection is that, with a close examination of an editor's commands, it can be seen that majority of the commands are translated to these three basic commands. Since these commands are directly operating on the text, the speed of the operations depends very much on the data structure employed. So one can realize the effect of the data structure on the performance of the editor. The efficiency of these commands will be used in this thesis as an important criterion for the evaluation of the data structures. From now on we will sometimes refer to these commands as 'primitive commands' or briefly 'primitives'.

2.1 BATCH EDITING SYSTEMS

Computerized editing started with the early non-interactive editing systems. The earliest form of editing was the manipulation of punched cards. The user had all his text (program) in a punched card deck. The corrections were made by retyping the mistyped cards, one by one. In each card, the user had to retype the incorrect characters and duplicate the rest. If the insertion of a new word caused overflow in the card, a new card had to be

inserted in the deck to handle the overflow. After any single change in the text, the whole card deck had to be re-read by the card reader.

'Batch editing' systems were created in the early 1960's to solve the problems associated with punched cards. Here, the user's card deck was stored on a tape or disk as a card-image file. Each card was referenced by a unique reference number. Corrections were made by running the 'edit deck', which contained very elementary editing commands, through the batch editor. The input file was read sequentially, and written in an output file after modifications. The commands that referred to line numbers in the text had to be sorted in ascending order of the line numbers within the edit deck. Since the changes were normally done to only a small proportion of the file, these kinds of editors spent most of their time in copying the text from one file to another.

One of the editors which performed batch editing (in off-line mode) was 'EDIT' developed at the University of Cambridge (Hazel 1974). Files were transferred to and from the editor via input and output 'streams'. The file from input devices such as card, tape, or paper-tape was stored on disk by the operating system. Two input streams were presented to the editor: one was the text to be edited, and the other contained the editing commands. The commands had to be in ascending order of line numbers (although it was also possible to read in and sort all the editing commands before applying them to the text). During the editing only one line of text was in memory at any time, so there was a

disk input (and possibly an output) call for every line. The commands available in this editor were the elementary forms of the basic commands insert, delete, and replace. After editing the file, it was stored in an output stream. It was necessary to produce a line-numbered listing of the text after each editing, so that the line numbers were in the most updated form.

The development of true data structures starts with the development of interactive editors. The following sections give a detailed description of this development.

2.2 SEQUENTIAL EDITING SYSTEMS

The early interactive systems still processed the files in a sequential manner similar to the previous batch editors. So there were the same limitations as with the batch editors. Normally, there was one input and one output file. The command lines had to be entered from the terminal in ascending order of the line numbers. So only forward motion in the text was allowed. Most of the editing time in these editors also was spent in copying the text from one file to the other.

One of the typical editors of this kind was an improved version of 'EDIT' designed for an IBM 370/165 (Hazel 1974). The lines of text were read line by line, under the control of the editing commands, which were also read line by line. After each line was processed it was sent to the output file. So no backward motion within the file was permitted. It was possible to switch between several files. This enabled the files to be merged, or parts of the text to be

moved from one place to another.

The 'ZED' Editor, developed at the University of Cambridge (Hazel 1980), allowed a limited backward motion within the file by keeping some of the previous lines in the main memory as long as possible before writing them in the output file. The backward distance that could be accessed depended on the buffer area used for the file.

To avoid the unnecessary copying of text from one file to another, some sequential editors used only one file for editing. One such editor was designed for a CDC 6600 at the Institute for Defence Analyses (Irons and Djorup 1972). In this editor, all the editing commands operated on the original copy of the file. It was possible to move both forward and backward within the file. With this method the insert and delete commands were very slow, since the file had to be expanded or contracted. For this, the editor relied on the fast copying capabilities of the CDC 6600.

Another editor called 'SITAR' (Schneider and Watts 1977), implemented on a PDP/11, used a slightly different structure to improve the insert and delete commands. The file contained an expansion area or 'hole' to take care of the insertions and deletions. As the user moved through the file (forward or backward), the hole moved with him. This was done by taking each record from one side of the hole and writing it at the other side. The new insertions were entered in this hole. Accessing the file was still sequential in this structure.

2.3 DIRECT EDITING SYSTEMS

The editors studied above were simple editors that did not need any special structure for their work files. The editing commands directly modified the original file in a sequential manner. With the development of more advanced interactive editors, it became necessary to use a copy of the file with a special structure suitable for the editing commands. Probably the first editor which used a different copy of the file to be edited was the original version of the well known 'QED' Editor, developed at the University of California at Berkley (Deutsch and Lampson 1967). In this editor, the whole file is copied into the main memory and is considered as a large string of characters. The only structure imposed in this string is by storing and interpreting the carriage returns as line delimiters. There are only three pointers showing the state of the current text buffer: one to the start of the first line, one to the end of the last line, and one to the start of the current line.

With this structure the primitive commands are very slow, due to the extensive copying of characters, or a large number of tests: To access line 'i', all the characters, starting from the top of the file, are examined, and the carriage returns are counted. The characters between the 'i-1'th and 'i'th carriage returns, including the latter, are the desired line text. To delete a character string, all the characters to the right of the string are shifted left, and overwrite the old characters (from now on, this procedure will be called 'contraction'). To insert a

character string, all the characters to be inserted are stored in temporary storage. When they are completely typed in, all the characters after the point of insertion are shifted right, far enough to make room for the new string, which is copied into the space created for it (this procedure will be called 'expansion').

As can be seen, the speed of these commands is dependent on two factors: the place in the text where the operation is done, and the size of the file being edited. It can be shown that for a file of 'N' characters, the average number of shifts, for single delete or insert commands at random positions, is $N/2$. For this reason the performance of the editor is decreased rapidly with an increase in the file size. Another major disadvantage of this structure is that the size of the file to be edited is limited to the size of the available main memory.

Following is the description of some other editors which use a variation of the above structure to overcome the disadvantages.

In a later version of the QED Editor, the designers introduced a different structure in which the large text is divided into a number of large segments, leaving some free space at the end of every segment. So the contraction and expansion takes place in one segment only for small deletions and insertions. But, since the segments are organized sequentially, for large deletions and insertions it may be necessary to reorganize the segments again, which is a time consuming operation. The designers also realized a second advantage of dividing the text into segments, that

most of the text can be kept out of the main memory most of the time. But again, since the segments are organized sequentially, there may be a large amount of disk i/o for some operations.

The next editor described here is 'QUIDS' developed at the university of London (Coulouris et al 1976). This editor basically uses the same structure as QED, with some extra control information stored with the text. QUIDS is in fact a document editing system. so the basic unit of the stored text is a paragraph rather than a line. The whole text is read into the main memory at the beginning of the editing session. The maximum buffer size for the text is 16000 characters. There are simple facilities for the segmentation of larger files.

The text and other information are stored as 'items'. There are three basic types of items: paragraphs, which are the basic units of text manipulated by the editor, format specifications, which affect the layout of the paragraphs, and non-sequential text, such as page titles and footnotes. Each item is stored with three pieces of control information: the 'item type', the 'countability code', which is used to assist in computing the item numbers, and the 'record length', which makes the sequential processing of text relatively easy. With this structure, although sequential access is done faster by making use of the record length, the insert and delete commands are still inefficient, i.e the whole text after the point of operation is expanded or contracted. The operations become more inefficient for large files.

The 'CMS' Editor, designed for an IBM/360 (Rice and Van Dam 1972), makes use of the virtual memory facilities of its host operating system. At the beginning of the editing session, the whole text is copied into the (virtual) main memory, so the maximum file size is limited by the maximum virtual memory size of an IBM/360. The lines are of fixed length, and they are linked together (forward and backward links) by absolute virtual address pointers. Access to a line is done sequentially by following these pointers. The delete and insert operations are more efficient in this structure, but after doing some editing operations, accessing the text lines may become a very slow operation, due to the numerous i/o activities. For example, if logically contiguous lines are not kept close enough to each other in the virtual memory, this will increase the number of i/o transfers required for sequential line accesses.

S. Pramanik and E. T. Irons suggest a data handling algorithm for on-line editors which is an attempt to solve the problems mentioned for the above data structures (Pramanik and Irons 1979). This structure utilizes a text which consists of several substrings. Initially the text consists of a number of substrings, each fitting into a fixed length page. these substrings are stored on the blocks of disk. The substrings are logically connected through a 'map'. Each map entry has two items: the address of the string, and the length of the string. The address is given by the page number and the offset of the substring within the page. Insertion and deletion are done by

manipulating the map as follows: for inserting a new character string into the text, it is directly moved into a main memory buffer, and is considered as a new substring. Then the entries in the map are updated. If the new insertion splits an existing substring, two entries are created in the map, otherwise only one entry is created. The former case, however, occurs most of the time. After an insertion, if the buffer in the main memory is filled, it is stored in disk storage. The following figure shows the necessary changes on the map, after inserting an string:

address	length
La	6

map for string 'ABCDEF'

address	length
La	4
Lx	3
La+4	2

after inserting XYZ (after D)

Figure 2.1

Changes on the map after inserting
an string

To delete a character string, only the entries in the map are updated, and there is no change in the actual text. If the string to be deleted lies between the first and the last characters of an existing substring (excluding both), a new entry is created on the map. This situation occurs most of the time.

It is clear that the map is continuously growing as more time is spent on editing. This is because of the substrings being split by the editing commands into more

substrings of smaller size. The following table shows the number of new entries created for each command, as a function of the point at which the operation is done (insertion is done preceding the character):

point of operation	insert	delete	text substrings
a	1	0	XXXX XXXXX XXXXX
b	2	1	^ ^ ^
c	2	0	
			a b c
			point of operation

Table I

Number of new entries created for
insert and delete

The designers of this structure compare the main memory requirements of the structure with the previous direct editing systems, in which the text is stored in the main memory as a continuous string of characters. For an average editing session of two hours, they show that in this data structure the map requires much less core storage than the direct editing systems. But as more time is spent on editing, the map grows larger. With the increase in the size of the map, the processing time for accessing a line or string increases, and hence the total performance of the editor is decreased. So there is a need for the substrings to be remapped, from time to time, into a continuous string to increase the editor's performance.

2.4 DATA STRUCTURES USING LINKED LIST OF PAGES

The next stage in the historical development of editor data structures started when the designers realized that to increase the efficiency of the editor, it is necessary to divide the whole text into several small 'pages' or 'blocks', and employ an elaborate and complicated mechanism for handling these pages. This would increase the efficiency of the primitive commands, and also decrease the number of disk i/o operations. For example, the access command becomes a more complicated operation: the designer must consider the access to pages in the secondary store, as well as to the lines or strings. Although some of the data structures described earlier used some sort of page structure, these pages were stored sequentially on disk file and did not have any logical relation to each other such as links.

Two of the earliest editors which used a linked list structure of pages were ATS and VIPCOM (an improved version of ATS), developed by the IBM Corporation (Rice and Van Dam 1972). In these editors the text is divided into fixed size pages (256 characters). Each page is linked to its neighbours by two pointers, one forward and one backward. These pointers are absolute disk addresses. There is also other control information in each page, such as the length of the text stored in the page. In the rest of the page the actual text resides, which contains a number of variable length lines (at least one). There is also some amount of free space in each page reserved for future expansions. The

lines within the page are separated by carriage returns, and they are not split between pages. The following figure shows the format of a page:

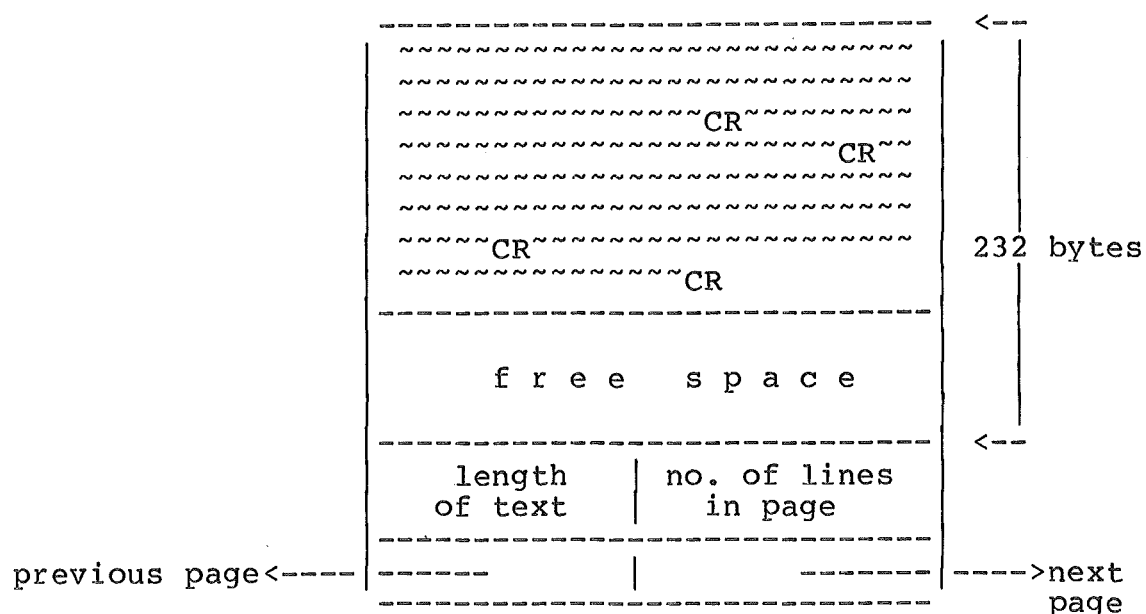


Figure 2.2

Structure of a page in ATS Editor

In this system a line is accessed sequentially by starting either from the top of the file or from the current line location, and following the linked list of pages (using forward or backward pointers and the 'number of lines in page' field). Within a page the line location is found by counting the number of carriage returns. Accessing a line in this manner may involve several disk reads if the main memory buffer used for the file is not large enough.

The operation of the insert and delete commands is improved, since expanding and contracting the text is now done within a single page. This is the main reason for keeping a free space at the end of each page. The expansion

of text within a page is only done when a line is altered. If the altered line would cause the page to overflow, a new page is created and linked to the other pages. To insert a new line, it is stored in a new page and this page is linked with the other pages in the desired location. This may split an existing page, where the new line is to be inserted, into two pages, creating a total of two new pages for a single insert.

Although this mechanism speeds up the operation of insert and delete commands, however, there are other problems. One of the main problems is that as more editing is done, the free spaces in the pages are increased and hence the number of pages become greater. As an example, consider the movement of several lines from one place to another in the text. Suppose the lines to be moved are in page 2 and in the parts of page 1 and 3 (labeled 1.2 and 3.1, in figure 4). They are to be moved to the middle of page N. For this, each of the pages 1, 3, and N will be divided into two individual pages, and linked to each other as shown in figure 4. So three new pages are created by this command, which includes no deletion or insertion of new text:

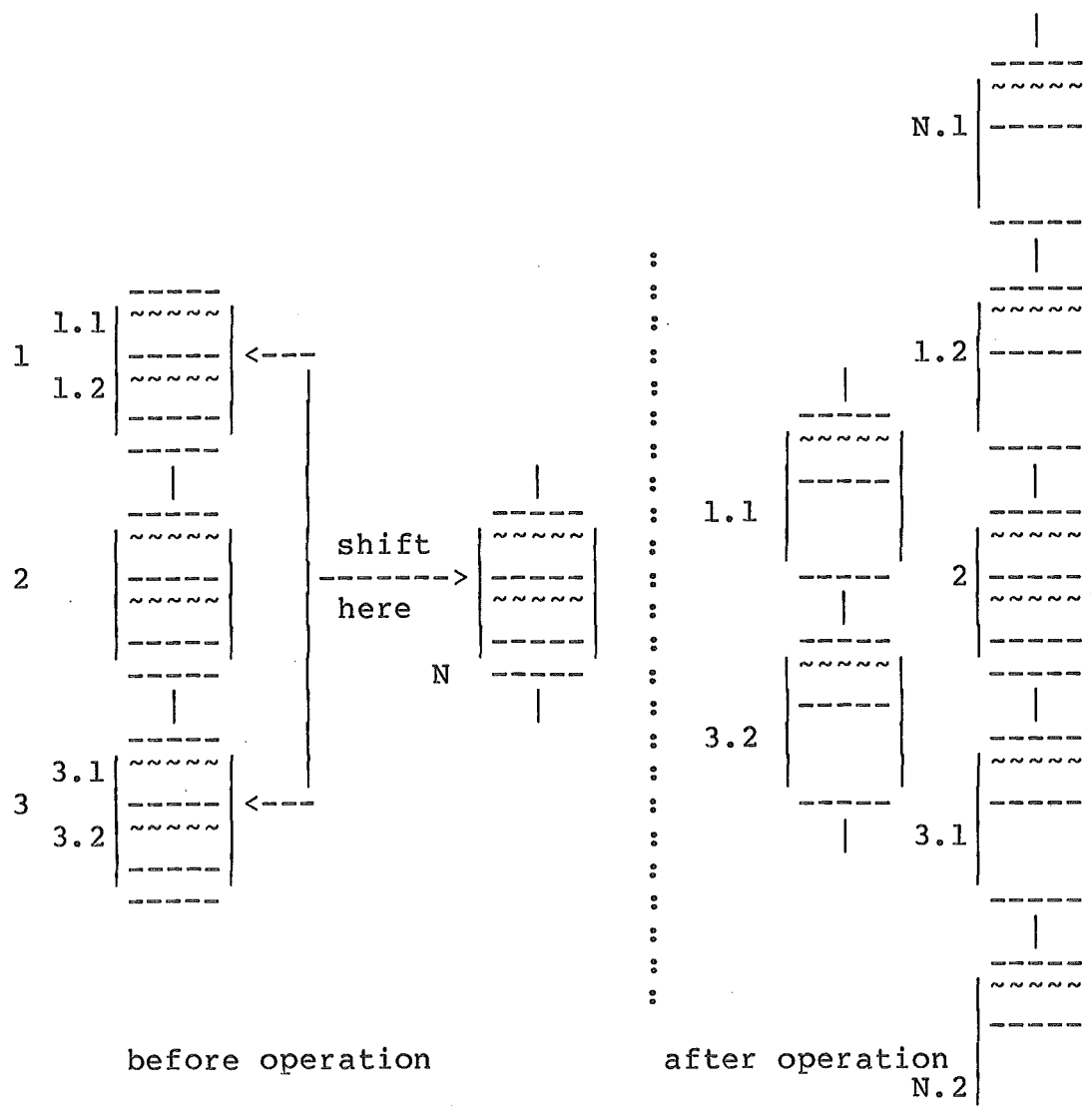


Figure 2.3

Movement of lines in ATS Editor

The solution to this problem adopted by the designers of the ATS Editor is that the system compacts the pages, from time to time, by moving them to a new file.

Another editor which uses a linked list of pages was developed at Brandeis University (Benjamin 1972). The design objectives of this editor explicitly included having no limitation on the file size, and efficient operation of the primitive commands. This is the first editor where most

of the conventional file system functions are specially tailored and incorporated in the editor in order to create an efficient editing system for a small single user machine. The files are created and maintained by the editor itself. There are two sets of files in the system. For 'stable files', whose structure and contents are not being modified, the text is stored sequentially within the disk pages. The second types are 'temporary files'. A file becomes temporary after the first editing change is made and becomes stable again after the editing session.

Before any file is created, the system reserves a fixed page with three pointers to the following pages: one to the first page containing the list of commands (dictionary), one to the page containing the list of files (directory), and a last pointer to the free list of pages (see following figure).

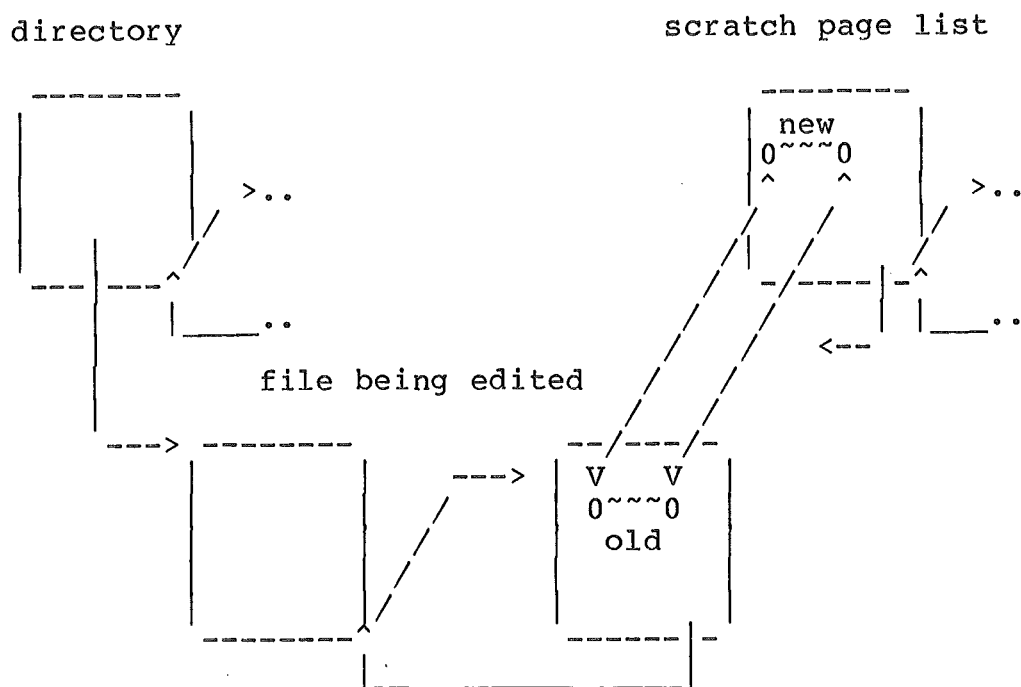


Figure 2.5

Structure of a file list during editing

The insert and delete commands are efficient with this structure, since extensive movement of text is eliminated, even within a page. But as modifications are made, the scratch list grows, and hence the sequential format of a stable file is lost. As a result, access to the lines becomes increasingly slow. To solve this problem, it is sometimes necessary to reconstruct this file, and change it to a stable file. This may be an expensive operation.

2.5 DATA STRUCTURES USING RANDOM ACCESS PAGES

The next development in editor data structures was to use an array of page pointers (page table) in the main memory, to access the pages randomly. One of the editors

using this structure is a multiuser editor called 'WYLBUR' developed at Stanford University (Fajman and Borgelt 1973). In this editor the text is stored in fixed size pages. Each page contains a variable number of characters. No linked list of pages is maintained by this editor. To manage the storage of the pages in secondary store, the editor depends on the underlying file system. But it gets a list of (absolute) disk addresses of the pages from the file system and keeps them in a table in main memory (called the work file directory). Each entry of this table contains two values: The first line number on the page, and the address of the page. So accessing a line is now done efficiently by searching this list and finding the the address of the page which contains the desired line. This page is read into memory and is searched sequentially for the line. In WYLBUR, each line is assigned a line number which serves as an identifier for the line. The numbers range from 0 to 9999.999, with three digits after the decimal point. When creating the text only the integer line numbers are used, and the fractional line numbers are kept for the later insertion of new lines.

To insert a line, the page containing the desired place for the insertion is read in, and the text within the page is expanded to create enough space for the new line. If the line can not fit into the existing page, a new page is acquired, and the old page is split into two. In this case the work file directory is also updated by expanding it. Deleting a line is done by contracting the text within the page.

With this structure the primitive commands, especially accessing a line, are implemented efficiently for small files. For large files the size of the directory is big, and since primitives operate on this directory (search, expansion, and contraction), the speed of these commands will be slow. The question of how much free space must be kept in each page for later insertions also remains unanswered by the designers.

David E. Rice and A. Van Dam present a theoretical discussion of editor data structures in their article, which is one of the few articles written on this subject (Rice and Van Dam 1972). They distinguish the following parts in a data structure:

- a. The external divisions which the user deals with, such as lines, superlines, paragraphs, etc.
- b. The internal divisions of the text (pages or blocks) for storage purposes in the main memory and secondary storage.
- c. The internal representation of the text and the techniques used for performing the primitive commands.

They compare the traditional program and data paging strategies with the paging for an editor work file. They realize the more dynamic and complex environment of an editor work file. Based on the traditional techniques, the authors develop new paging strategies for user working sets, placement and replacement algorithms, and the determination of the page size and the size of the main memory paging area for an editor.

For internal representation of the text, they

distinguish the following methods:

1. Text stored with no control information for structuring or formatting, i.e a continuous string of characters.
2. Text stored together with control information. For example when the size of the line or the pointer to other lines is stored with each line.

All the editors described so far are in the above two categories.

3. Text and control information are stored separately. In this case a pointer to the actual line is always a part of the control information. Other information may be the size of the line, the page address, and so on.

Editors which use the last method above have proven to be more efficient editors. In the rest of this chapter one such editor will be described. The CHEF editor, and the new data structure, explained in the next two chapters, use a combination of methods 2 and 3 above for the internal representation.

2.6 DATA STRUCTURES USING AN ARRAY OF LINE POINTERS

Probably the first editor which used an array of line pointers is an editor called 'GODOT' (Macleod 1977). In this editor the array is kept in main memory. There is one word entry in the array for each line of the text, which contains the address of the corresponding line relative to the top of the file. The text is stored in fixed length records on disk. Each record corresponds to one line, so a certain amount of space is wasted for each line, but it

makes the addressing and garbage collection easier (see figure 7).

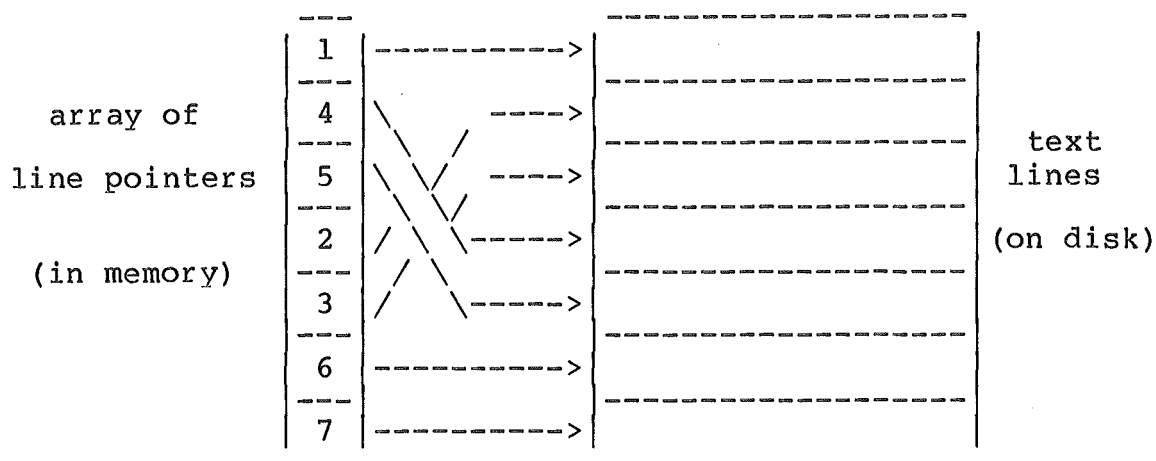


Figure 2.6

Workfile structure in GODOT Editor

This structure has a major impact on the efficiency of the primitive commands as follows:

To insert a new line, the actual line text is added to the end of the file, and a pointer is created at the corresponding position in the array of line pointers by expanding it.

To delete a line, only the array of line pointers is contracted. Access to any line is made by indexing the array entries. This allows random access to the desired line without any need for searching.

With this structure, the actual text is little affected, and most of the commands operate only on the array of line pointers. For example, the movement of lines within the work file, which can be a complicated and expensive operation in the previous structures, is simply performed by movements of the corresponding entries in the array. There

is no movement of the actual text for any command, and thus the text structure remains sequential and simple all the time. As a result, there is no need to reconstruct the text frequently, as was necessary in most of the previous structures.

The structure used in the GODOT editor is the simplest form of those using an array of line pointers. There are a number of disadvantages as follows:

1. Since the array is kept in the main memory, there is a limit to the size of the file to be edited. This is a particular disadvantage in small systems with static memory allocation.
2. Manipulation of the array (expansion and contraction) will be slow for large files.
3. Since there is a maximum length reserved for every line of text, a considerable amount of space is wasted.

Using an array of line pointers is probably one of the most important advances in the development of editor data structures. There are only a few editors which use this method. CHEF is one of such editors. The problems mentioned above for the GODOT editor are solved in CHEF by sophisticated techniques and novel ideas. The next chapter is a detailed description of this editor, which gives a basis for understanding the new data structure in chapter 4.

CHAPTER 3

THE CHEF EDITOR DATA STRUCTURE

In this chapter the CHEF editor data structure is described in detail. There are a number of reasons to assign a whole chapter to this data structure:

- i. The new data structure, presented in the next chapter, is based on the CHEF data structure.
- ii. CHEF has one of the most advanced data structures as mentioned in the historical development.
- iii. CHEF is to be used as the editing server of the Department of Computer Science network.

3.1 GENERAL DESCRIPTION OF THE DATA STRUCTURE

The CHEF work space data structure consists of two distinct parts:

- i. A linear array of line pointers, called the 'pointer-place', numbered from 1 to 'last_line'. Each entry of this array indicates the starting position of the corresponding text line.
- ii. A one-dimensional storage area for text, called the 'record-place', where the variable length lines are stored one after another without gaps. The size of each line is also stored with the line.

Conceptually, the lines are accessed randomly in this structure. The 'i'th line of text is accessed by first accessing the 'i'th element of the array of line pointers,

and then following the pointer to the text. The following figure shows this data structure with some of the variables indicating its current status:

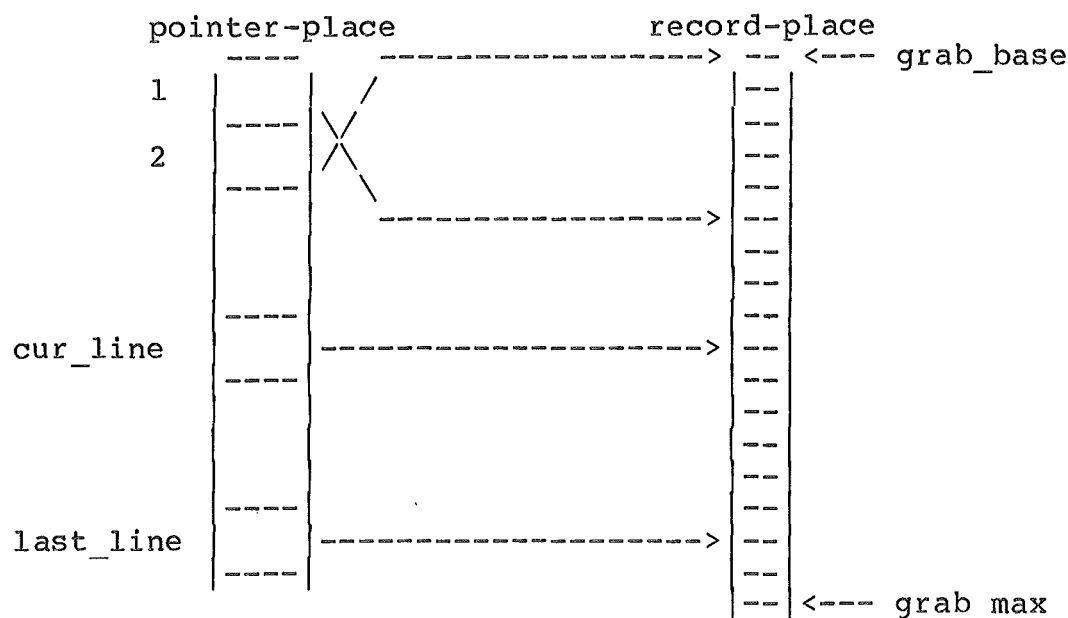


Figure 3.1

Conceptual view of the CHEF data structure

In CHEF, the actual text is stored on disk to raise the limit on the file size. There are two 'input' and 'output' buffers for the text in memory (in the original version of CHEF, there was only one buffer but by increasing it to two, the speed of certain commands was greatly increased). There are two factors associated with this structure which limit the size of the file to be edited. Following is a description of these factors, together with the solutions adopted in CHEF.

1. The first factor is the number of lines in the file. This is directly dependent on the size of the array of line pointers. In the other editors which use an array of line pointers (e.g. GODOT), the array is kept in main

memory. In this case a limit has to be put on the size of the array, which limits the number of lines allowed for the work file. To solve this problem, CHEF uses secondary storage for the pointer-place as well. There are only two blocks of pointer-place in the memory at any time. A simple virtual memory technique with a 'least recently used' algorithm is used for swapping the blocks in and out.

2. The second limiting factor is the number of characters in the file. Since in CHEF a line pointer indicates the starting character position of the line in the record-place, any limit to the bit size of the line pointer will create a limit to the character size of the file. The pointer is limited by the word length of the machine used. For example for a 16-bit computer, with two's complement representation, the limit is 32767. In CHEF, this problem is eased by storing the start of each text line in the record-place at discrete addresses spaced by a fixed number of bytes apart. This wastes a small number of bytes between lines, but it reduces the number of bits needed to specify a line pointer. The spacing can be chosen by the implementor to suit his computer. The following table shows the maximum allowable size of the file as a function of the maximum distance between line starts (for a 16-bit computer). It also shows the average number of bytes wasted for each line:

spacing (bytes)	max file size (K bytes)	ave bytes wasted per line
0	32	0
2	64	0.5
4	128	1.5
8	256	3.5

Table II

Maximum workfile sizes for different
line spacings

In the CHEF terminology, the number of bytes used for spacing is called a 'grab'. A grab size of 4 is used for most of the CHEF implementations on 16-bit computers, enabling a file of 128 K bytes to be addressed.

So far we have assumed two disk files, the 'record-store' for storing the actual text lines, and the 'pointer-store' for storing the pointers. There is a third disk file called the 'backup-store', which is used to store the pointers for undoing the commands (explained later in detail). For each pointer block there is a corresponding backup block. So the sizes of the pointer and backup files are equal. For accessing purposes, CHEF assumes that all the disk blocks of each of these files are stored as a sequence of contiguous blocks numbered 0,1,2,... To avoid the need for three files, all these files are merged into one. In this file, blocks of the three types are interleaved in a fixed ratio to maintain the original sequentiality of each file. This ratio is currently one

pointer block and one backup block to every 6 text blocks. So the numbering of the blocks of the merged file is as follows:

file blk.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
pointer blk.	0	1	2	.	.
backup blk.	0	1	2	.	.
text blk.		0	1	2	3	4	5				6	7	8	9	10	11			12

The mapping of the pointer block number (Bp), backup block number, and the text block number (Bt) onto the corresponding (merged) file block numbers (Fp, Fb, and Ft respectively) is done in CHEF using the following relations:

$$Fp = Bp * 8$$

$$Fb = Bp * 8 + 1$$

$$Ft = (Bt / 6) * 8 + Bt \text{ REM } 6 + 2$$

('REM' is BCPL operator corresponding to 'MOD' in Pascal)

3.2 THE IMPLEMENTATION OF THE MAIN COMMANDS

In this section, the implementation of the three primitive commands in CHEF is described in detail.

3.2.1 Accessing a Line

A line is accessed randomly through the array of line pointers. To access line 'i' of the work file, the pointer to this line must be accessed first. The pointer is in the 'i'th entry of the array, and the array is stored in the sequential file blocks of the 'pointer-store'. Since each block contains an equal number of pointers, the sequence number of the block containing the pointer to line 'i' is

calculated by:

$$Bp = (i + line_base) / block_csz$$

where 'line_base' is one less than the first line of the current work file, and 'block_csz' is the (fixed) number of pointers in each block section 3.3 discusses the need for the variable 'line_base').

The pointer block number obtained above (Bp) is mapped onto the file block number as shown in the previous section. The file block, which contains the desired line pointer, is then read into memory, if it is not already there.

The value of the pointer obtained in this way shows the starting position of the line within the record-place (in grabs). This value is converted into two values, a text block number (Bt) and the offset of the line within the block, by the following relations:

$$Bt = \text{pointer value} / block_gsz$$

$$\text{offset} = (\text{pointer value} \text{ REM } block_gsz) * 4$$

Where 'block_gsz' is the number of grabs in a disk block.

This text block number is mapped onto a file block number by the relations shown in the last section and the block is read into memory, if it is not already there. The desired line, which is stored in this block from the position shown by 'offset', is copied into the buffer 'line', and passed to the calling program. Accessing a line in CHEF is carried out by a procedure called 'fetch_line(i)'.

With this access method, the major factor affecting the speed of the operation is the number of i/o calls for reading and writing the blocks. There are between 0 to 2

calls for block 'read', and 0 or 1 call for block 'write'. The exact number depends on two factors: whether the desired block (pointer or text) is in memory, and whether the old block to be replaced has been changed since being read in (only for pointer blocks, since there are separate input and output buffers for the text blocks). This is shown in the following table:

block in memory?		old pointer block changed?	no. of calls		total
pointer	text		read	write	
Y	Y	X	0	0	0
Y	N	X	1	0	1
N	Y	N	1	0	1
N	Y	Y	1	1	2
N	N	N	2	0	2
N	N	Y	2	1	3

where: Y = Yes, N = No, X = no effect

Table III

Number of i/o calls for an access in CHEF

3.2.2 Insertion

To insert a new line between two existing lines of the work file, the only change in the record-place is that the new line text is stored at the end of it. The main operation is performed in the pointer-place. That is, the array of pointers is expanded to make room for the new pointer. This involves reading all the succeeding pointer blocks from the disk, shifting the pointers to the right within each block by one position, and writing them out on

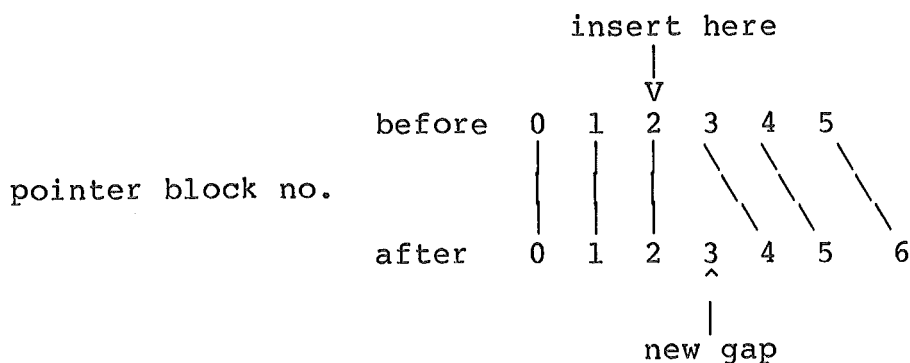


Figure 3.2

Creating a gap for an insertion in CHEF

When the new lines come from another region of the work file and the modifier is 'D' ('ID' command which is used for moving a group of lines within the workfile), the operation is performed as follows: the source lines are inserted in the new location and then they are deleted from the original place (by performing a delete operation as described below). So movement of lines is done by two separate operations of insert and delete.

3.2.3 Deletion

To delete one or more lines from the work file, since the number of lines to be deleted is known in advance, all the succeeding pointers in the pointer-place are shifted to the left (contracted), by an amount equal to the number of lines deleted. There is no change in the 'record-place', and the text of the deleted lines is simply abandoned. For contracting the pointer-place, all the pointer blocks from the point of delete command must be read into memory and modified.

The efficiency of the delete command has had some

influence on the implementation of the pointer-place. In the original version of CHEF, with the realization that most editing operations are sequential or within a small region of the work file, there was only one 'window' of the pointer-place in the memory, that could move up or down the array of pointers as required. This mechanism worked satisfactorily for most of the editing operations. But when a large number of pointers had to be moved a distance greater than the width of the window, it became very slow. One example of this situation is when the editor is required to delete the first 600 lines of a 1000-line file. For this, the pointers in positions 601 to 1000 must be moved to the positions 1 to 400. If the width of the window is less than 600 lines, the window must move up to fetch one pointer and down again to store it. This must be repeated 400 times, which requires at least 400 calls for block reads and 400 calls for block writes.

To increase the efficiency in this situation, two windows of the pointer-place are kept in memory at any time. These windows do not necessarily contain two consecutive pointer blocks, and they may be separated by an arbitrary number of blocks. As was said before, a simple form of 'least recently used' algorithm is used for swapping the windows in and out, as follows: when a particular pointer block is required, the two blocks (windows) in the memory are examined. If neither of them is the required block, the block that was not used most recently is replaced with the block required. There is a 'flag' associated with each resident block which indicates whether the contents of the

block have been changed since it has been in memory. If they have, it must be written on disk before being replaced, otherwise it is simply ignored.

3.3 THE IMPLEMENTATION OF OTHER IMPORTANT COMMANDS

In this section, the implementation of the commands that affect the data structure is described. These include: Undo, Replace, Execute, Justify, Tag, Change, New, and the control lines. As will be seen, in the implementation of these commands they are translated to the primitive commands access, insert, and delete. It is also important to realize that in CHEF, since the changed lines (replaced, tagged, justified, etc.) are always stored at the end of the record-place, the original lines are never destroyed and can be recovered by the mechanism explained below for the Undo command.

a. Undo Command (U): For undoing a command in CHEF, the inverse of the command is computed for those commands that only move or copy the line pointers, and saved in a buffer called 'trail_line'. If the next command is 'U', the inverse command is executed. For example, the inverse of '4ID7,8' is '8ID5,6'. For the commands that destroy the line pointers, such as 'D' (delete), or 'R' (replace), the lost pointers are saved in the appropriate 'backup block' mentioned before. In this case a new command, unavailable to the user, called 'B' (back substitute), is stored in the 'trail_line'. If this command is executed, it moves (if no modifier), or inserts (if modifier is 'I') the pointers from

the backup-store to their original positions in the pointer-store. The commands in CHEF are classified with respect to U into 3 categories: a 'willing' command modifies the workspace and can be reversed with 'undo'. An 'unwilling' command can not be undone. a 'neutral' command is used merely to examine the workspace and any number of these can follow a willing command before it is undone.

b. Replace Command (R): The replacement of a substring of a line by a new string is done by the 'R' command. The implementation of this command is done in the following order:

i. The specified line is accessed through the procedure 'fetch_line'.

ii. The line text is searched for the pattern. If the pattern is found, it is replaced by the new string.

iii. The altered line is stored at the end of the record-place, and its old pointer in the array is simply replaced by the new pointer.

c. Execute Command (X): The implementation of this command proceeds as follows: All lines of the desired region are accessed one by one and, if they contain the specified pattern, they are 'flagged' by negating their pointers in the array. In a second inspection of the region, the pointers are accessed and the remainder of the command line is executed for each of the flagged lines. The pointers of the flagged lines are finally restored to their original form by negating them for a second time.

d. Justify Command (J): For implementing the 'J' command, the old text lines to be justified are accessed one by one, justified, and stored at the end of the record-place. The pointers to these lines are inserted in the pointer-place, after the last line of the region being justified. At the end, the old line pointers are deleted.

e. Tag Command (T): It is possible to tag one or more work file lines with a single character, so that they can be picked out later. This is done by the 'T' command. To implement this command there is an extra character, called 'tag_char', associated with each line text in the record-place. When a line is tagged its text is accessed, the specified tag character is placed, and the line is stored at the end of the record-place. As with the replace command, the old pointer in the pointer-place is replaced with the new one.

f. Change Command (C): The 'C' command is used to replace one or more lines with new lines either from the terminal (if there is no modifier and no operand), another region of the work file (if there is a 'D' modifier or an operand), or another file (if there is a 'F' modifier). When the modifier is 'D', the source region lines are deleted after the operation. The implementation of this command is done by translating it directly to delete and insert commands.

g. New Workspace Command (N): This command is used for stacking the current work space. For this, the

important status information of the work space, such as 'grab_base', 'last_line', and 'cur_line' are stored in a special record at the end of the record-place. Stacking the pointer-place is done by adding the value of 'last_line' to a variable called 'line_base' (initially zero). In the new work file, access to line 'i' is performed by first increasing 'i' by the value of 'line_base'. The layout of the pointer-place is indicated in the following figure:

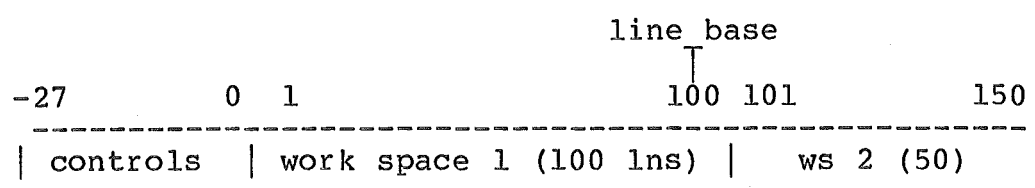


Figure 3.3

Layout of the pointer place in CHEF

On unstacking the current work space, all the space used for the line pointers and text for this work space is returned to the system by restoring the status information to its previous values.

h. The Control Lines: CHEF has 27 text buffers, called 'control lines', which are used for storing frequently used text lines or commands. These are given permanent positions at the beginning of the record-place. Each control line has the maximum possible length, which is the same as the maximum for an ordinary text line. When a control line is changed, the new contents are stored in the same place as the old one. So alterations to the control lines cannot be undone. The pointers to the control lines are stored in negative entries of the pointer-place, so that

they can be addressed as lines -1 to -27. For the actual implementation, a memory resident array of 27 cells is used for the control line pointers.

3.4 THE INTERFACE WITH THE UNDERLYING FILE SYSTEM

Since CHEF is primarily studied in this thesis as the basic editor of the editing server, it is necessary to study its interface with the underlying file system. This will help the understanding of the underlying file structure and its relationship with the CHEF data structure, in order to be able to design a single data structure at the editor level with the same overall effect. It also helps to highlight the disadvantages of implementing an editor data structure on top of a file system.

An editor is in fact a piece of utility software. It depends on the host system for providing facilities such as a multiuser environment and disk space management. The latter is performed by the file system. Ordinary editors are particularly dependent on this component, i.e. their data structure is built on top of the file system. They call the file system, whenever required, to create and manipulate the work files. This makes the logical design and implementation of an editor data structure easier, but it may produce considerable overheads in terms of disk input and output. It also creates inefficiency in main memory usage, since the file system utilizes its own internal buffers for i/o to files.

It was mentioned earlier that CHEF maintains an ordered sequence of consecutive blocks numbered 0, 1, 2,... for its

work space. These ordered blocks are mapped onto the actual random disk blocks by the file system. The mapping is done differently depending on the underlying file system and the file organization used for the work space. The file organization is selected by the CHEF implementor from one of those made available by the operating system. The following is a description of two systems at the University of Canterbury on which CHEF was implemented.

a. On the PRIME 750 computer, CHEF uses a DAM (Direct Access Method) file organization for its work space. With this organization, the system creates a tree structure of disk blocks for the storage and maintenance of user files. The actual user 'data blocks' (i.e. the CHEF work space blocks) are stored in random disk blocks, and there are one or more levels of 'index blocks' pointing to these blocks. This is shown in the following diagram:

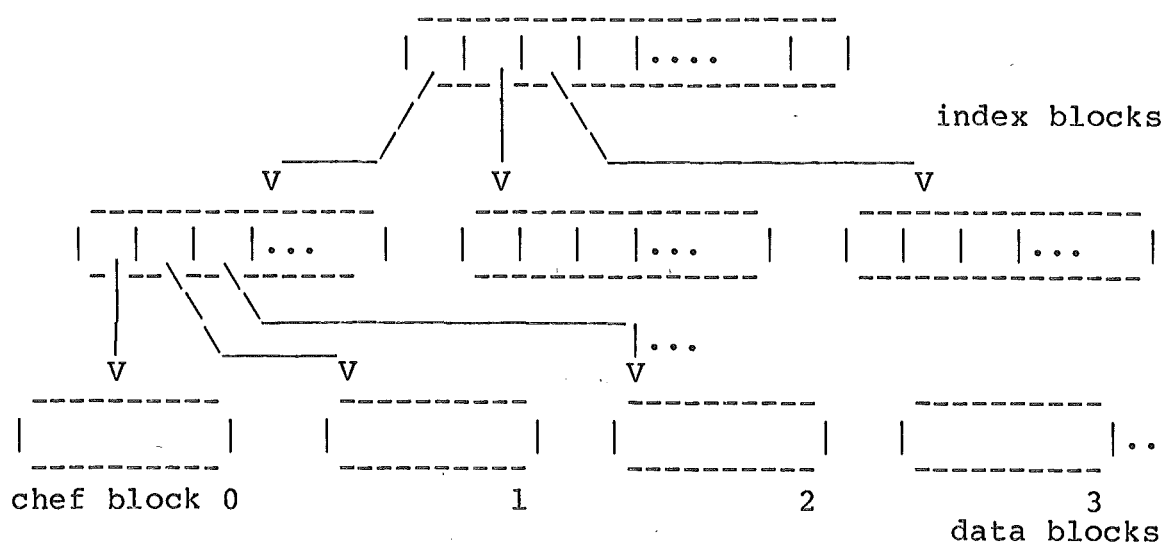


Figure 3.4

Structure of a DAM file on the PRIME 750

b. The second system that CHEF is implemented on is

the ECLIPSE S/130. In this system, CHEF uses an RDOS 'random file' organization. Here the user's 'data blocks' are stored in random disk blocks, and there is a linked list of 'index blocks' containing pointers to these blocks, as the following diagram shows:

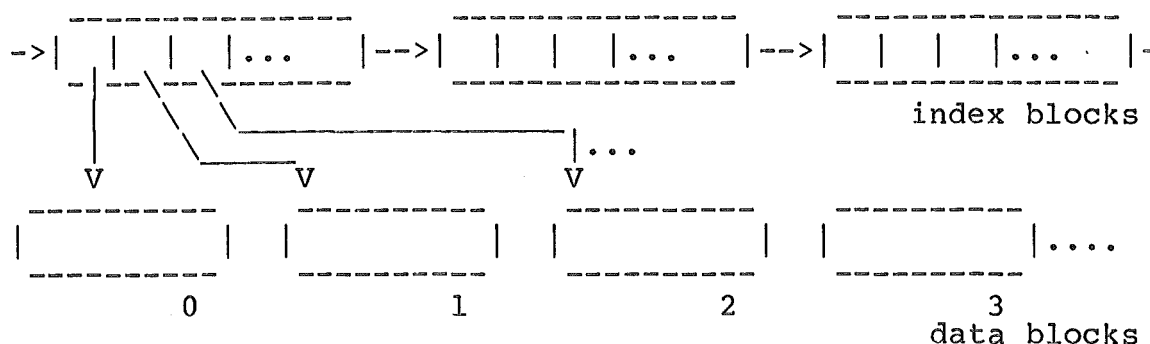


Figure 3.5

Structure of Random file on the ECLIPSE S/130

From the above description, it can be seen that a block 'read' or 'write' call performed at the CHEF level is in fact a call to the file system to traverse its own blocks structure (e.g. a tree or linked list) to read or write the required block. This creates considerable overheads for the operation of the main editing commands. To demonstrate this, the actual implementation of these commands is described in the following paragraphs for the version of CHEF implemented on the ECLIPSE.

a. Accessing a Line

Accessing a line proceeds in the following order:

1. The line is mapped onto a 'pointer block' number, and from that onto a CHEF work space block number by the simple calculations explained earlier.

2. If the pointer block is not in one of the two windows in memory, the RDOS file system is called to read it in.
3. The file system follows its linked list of index blocks to find the index block containing the actual address of the required block.
4. The block is accessed (a CHEF pointer block), and passed to the CHEF buffer area (one of the two windows).
5. The appropriate pointer to the desired line is retrieved from the pointer block. The pointer gives a sequential text block number and the offset of the line within the block. The text block number is converted to a work space block number, as shown earlier.
6. The file system is called again to retrieve the desired block. For a second time, the file system follows its linked list of index blocks to access the text block.
7. The block is read and passed to the CHEF input buffer for text.

b. Insertion

The insertion of a line (or lines), which may involve a large number of disk i/o operations, is carried out in the following order:

1. All the steps for accessing a line are performed to store the new line text in the work space.
2. To create a gap of one block in the pointer-place, all the pointer blocks after the point of insertion are read, renumbered , and written out in their new

positions. For each 'read' and 'write' carried out, steps 2, 3, and 4 for 'access' must be repeated.

3. After inserting the new pointer(s) in the gap, the gap is closed up which involves accessing all disk blocks containing the tail portion of the pointer-place. As above, steps 2, 3, and 4 are repeated for each block 'read' and 'write'.

c. Deletion

To delete a line pointer, all the pointers after the point of operation are read, shifted to the left, and written out on disk. Again this requires steps 2, 3, and 4 to be repeated for each read or write.

The exact number of disk i/o operations for performing the main commands depends on the following factors: the size of the workfile, the 'distance' of the current line from the desired line, the number of file system internal buffers and whether the operating system maintains a 'cache' of blocks, and if a linked list is used (e.g. on ECLIPSE) whether the index blocks can be traversed both forward and backward.

These were taken into account in the design of the new data structure described in the next chapter.

CHAPTER 4

A NEW EDITOR DATA STRUCTURE

In this chapter, the basic concepts of the new data structure are described.

4.1 GENERAL CHARACTERISTICS

As shown in chapter 2, there was a logical development of editor data structures. In the direct editing systems, the whole text was stored in memory and the operations directly affected the text. It was later decided to store the text on several disk pages (blocks) to be able to edit larger files with possibly better performance. The early page structures utilized the pages sequentially. Later, a linked list structure was used for the pages. It was then realized that, by keeping the page addresses in memory, the performance could be improved. With the development of data structures which utilize an array of line pointers, a new phase started in which the text was stored on disk and remained sequential and unchanged. All the editing commands were now affecting only the array, which was much smaller than the text itself. In this type of structure, since the commands affect the array in the same way they affect the text (expansion, contraction, etc.), it is logical to think that the same development will be repeated for the array of line pointers. This new phase starts with the whole array kept in memory (e.g. GODOT). CHEF introduced a page

structure for the pointer array as well as the text, and stored most of it on disk. On CHEF, the array pages (pointer blocks) are utilized sequentially. one would expect that a linked list structure of the pointer blocks would have a better performance and this structure was suggested by Dr. M. A. Maclean in 1982. It was later decided that further improvements were possible by utilization of a memory resident table for the access and manipulation of the pointer blocks. This further shifts the operation of the main commands from the array to a much smaller resident table. This makes the new data structure inherently more efficient for editing purposes than the earlier structures studied in chapter 2.

Another advantage of the new data structure is that it is capable of handling its own dedicated disk space for several users' work files. This makes it possible for the editor (CHEF) to be implemented in the editing server without the need for a file system. And in this case, since there is no underlying file structure, a higher performance is expected for the editor.

The new data structure is designed to be used in two different environments: In an editing server with a dedicated disk, and on top of a file system. There are small differences in the data structure for different environments. These differences will be pointed out, whenever required, in the following sections.

4.2 GENERAL STRUCTURE

The new data - structure consists of at most four components as follows:

1. An internal table, called the pointer block 'directory', used for accessing and manipulation of the pointer blocks. Each entry of the directory has two values: The address of the corresponding pointer block, and the number of pointers in the block.
2. The array of line pointers, stored in randomly allocated blocks of disk (in a dedicated environment), or a file (in a file system environment). Each pointer block contains a variable number of pointers (between reasonable bounds) to allow for efficient insertion and deletion.
3. An optional memory resident 'map-table' used for accessing the text blocks. This table is only required when the data structure is used in an editing server on a 16-bit computer.
4. The actual text, stored in randomly allocated blocks of disk or a file. The format of the lines stored in the blocks is basically the same as in CHEF. All the new or altered lines are stored at the end of the record-place (in the last text block).

These four components are organized in a hierarchical structure. The highest level of the hierarchy is the directory, and the text blocks are at the lowest level. There are at most two levels of indirection for accessing the text: First the required pointer block (and the offset

of the pointer within the block) is accessed through the directory, and then the desired text block (and the offset of the line within the block) is accessed through the pointer, either directly or through the internal map-table.

The same virtual memory technique used by CHEF is used for both the pointer and text blocks. There are two windows of the pointer-place in memory. A 'least recently used' mechanism is used for swapping the blocks in and out. There are also two input and output buffers for the text blocks. The following figure shows a conceptual view of the new data structure:

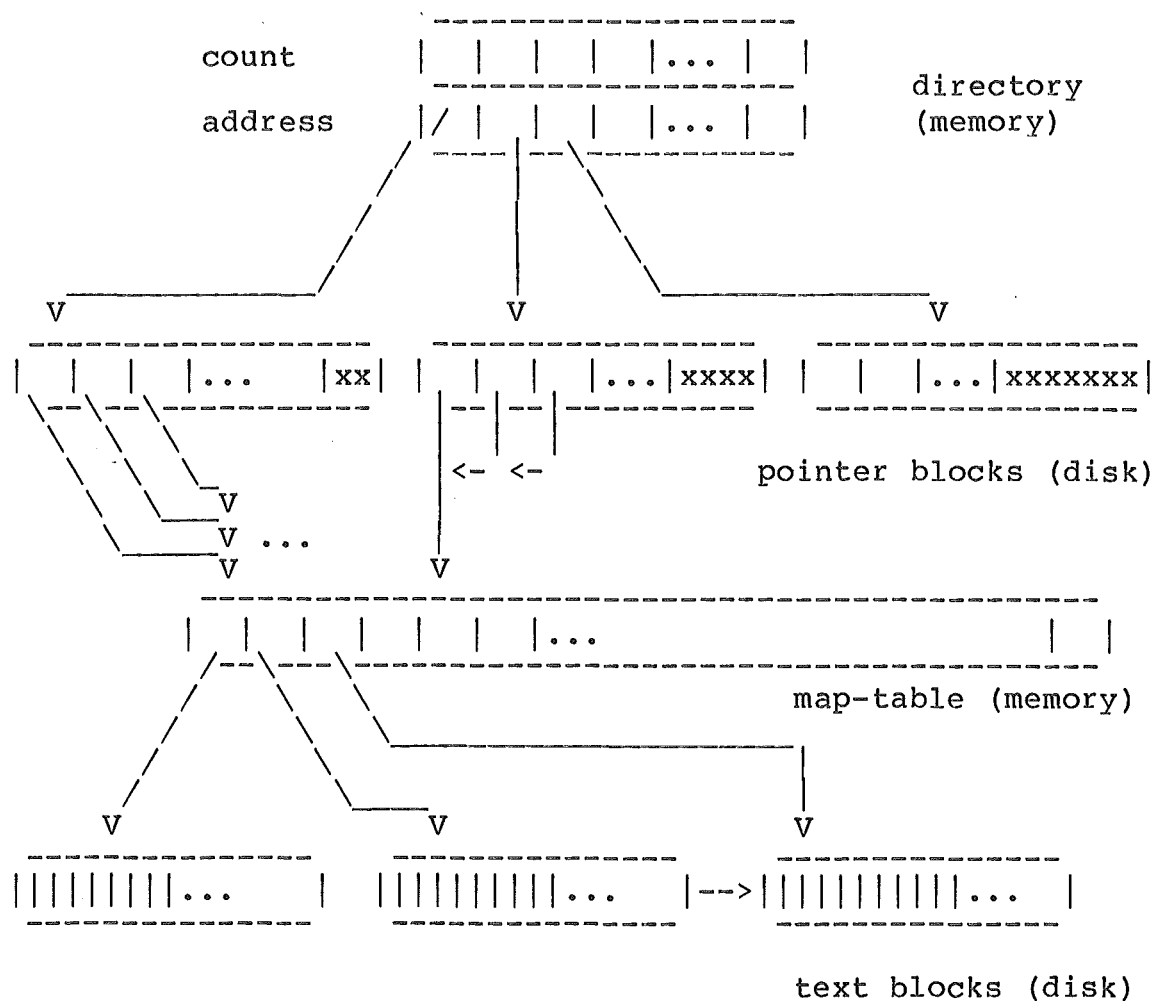


Figure 4.1

Conceptual view of the new data structure

With the pointer and text blocks stored randomly on disk and accessed via pointers, the only required function for the disk manager is the allocation and deallocation of the disk blocks. This can be done with simple mechanisms such as a 'bit map'. As with CHEF, the maximum file size is limited by the number of bits in a pointer. The resident tables may also impose some limitations on the file size, but these limits are reasonably large, and can be increased considerably by a small increase in the size of the tables.

4.3 DETAILED DESCRIPTION OF THE DATA STRUCTURE

The four components of this data structure are further elaborated in this section.

1. The Pointer Block Directory: The directory is the most important component of the data structure for two reasons:

- i. It makes it possible to store the pointers in random disk blocks.
- ii. It is the main component increasing the efficiency of insert and delete commands. This is done in two ways. First, by avoiding extensive movement of the pointers, since most of the operations performed on the array of line pointers are now done on the directory which has a much smaller size and can always be kept in memory. Second, by avoiding a large number of disk i/o operations for manipulation of the pointer blocks.

There is a two-word entry for each pointer block in the directory containing the address of the block and the number of pointers in the block. As an example of the size of the directory, if it is assumed that each pointer block is set to contains an initial value of 218 line pointers (e.g. on ECLIPSE S/130), for a file with 1000 lines there will only be 5 entries in the directory. This ensures the efficiency of operations such as insertion and deletion of the entries. With this structure, the maximum number of file lines is only dependent on the size of the directory. If the directory has 200 entries (2 words each), the maximum number of lines allowed for the file will be about 43600, assuming

a block size of 256 words and an average of 218 pointers per block.

2. The Pointer Blocks: The pointer blocks contain a variable number of pointers each. At the beginning of editing, when the blocks are created, an initial number of pointers is stored in each block. This value is optional and can be set by the implementor. Although it has no major effect on the performance, a value between 75 to 85 percent of the total capacity is recommended. As editing proceeds, this allocation of the pointers is changed, but there are always a minimum and maximum number of pointers that can be stored in each block. These values are commonly set to 100 (for minimum) of the total capacity. As an example, for the ECLIPSE Computer where a block size of 256 words is used, the values are 128, 218 (minimum, average, and maximum respectively). The experimental results (presented in chapter 6) show that these values are satisfactory.

Within a pointer block every line pointer is stored in one word. We must be able to keep two values in this word: the address of the text block, and the offset of the line within the block. The exact number of bits in each word allocated for each of these values will depend on the environment. For example, if in a computer a block size of 512 bytes is used, at most 9 bits can be used for indicating the offset of the line (with a grab size of 1). This number can be reduced, if necessary, by increasing the grab size. Out of the remaining bits in each word, one bit is always used for the 'X' command (for negating the pointers), and

the rest can be allocated for addressing the text blocks. For 32-bit and larger computers this problem disappears for all practical purposes, since there are at least 22 bits available for addressing the text blocks. This can address a disk space of up to 2000 Megabytes (a block size of 512 bytes is assumed). For a 16-bit computer, there are fewer bits available for addressing the text blocks. In a dedicated disk environment, it becomes necessary to address them indirectly through the internal 'map-table'. This will be further described in the next sub-section.

3. The map-table: As mentioned, this map is only used when the data structure is implemented in a 16-bit computer with a dedicated disk. The bit allocation of a line pointer in a 16-bit computer is as follows:

- one bit for the operation of the 'X' command.
- 6 bits for addressing the offset of the line within the text block.
- the remaining 9 bits for addressing the text blocks.

An address space of 512 blocks can be addressed by 9 bits. This might be enough when there is a file system which maps the sequential block numbers onto the actual disk blocks, but obviously it is not sufficient when a dedicated disk is used. So the map-table is used in the latter case, to increase the address space. The size of this map depends on the number of bits allocated for the text block pointer. This pointer points to an entry of the map, in which the actual disk block number is stored. This makes 16 bits available for addressing the disk blocks, which gives an

address space of 64K blocks. For a block size of 512 bytes, this is equivalent to a disk with over 33 Megabytes of storage.

With this configuration, since 9 bits can only address 512 map-table entries, there is a limit of about 260,000 (512*512) characters imposed on the file size. This is reasonably large, but it can be increased, if necessary, by changing the bit allocation of the pointer. The following table shows the changes in the maximum file size and the size of the map-table as a function of different bit allocations:

bit allocation block : offset	max file size (bytes)	map_table size (words)
7 : 8	65,000	128
8 : 7	130,000	256
9 : 6	260,000	512
10 : 5	520,000	1024

Table IV

A pointer bit allocations and its effects
in file size and map_table size

For 32-bit and larger computers, the limit on the character size of the file is very large. There is no need for the map_table in these computers.

4. The Text Blocks: As in CHEF, the text blocks are little affected by the editing operations. The only operations on these blocks are accessing existing lines, and storing new or altered lines in the last text block. There

are two buffers in memory for these operations:

- i. An 'input-buffer', for accessing the blocks. This buffer never needs to be written out on disk, since it is only used for accessing lines. The only exception is when a control line is accessed and altered, in which case it is written out in the original place directly from the input-buffer. A variable always indicates which text block is in the input-buffer.
- ii. An 'output-buffer' used for storing new or altered text lines. As in CHEF, a line is always stored at the end of the record-place, so the output-buffer always contains the last block of the record-place. Any new or altered line is written in this buffer from the first free location, pointed to by a variable called 'out_buf_off'. Whenever this buffer is filled, it is written out on disk in a newly allocated block, and hence it is free for accepting more lines.

It was shown that the text pointer gives the address of the start of the line within the block. For the lines that reside on two blocks, there should be a method to access the next block. In the CHEF data structure, since the blocks are utilized sequentially, this is done by done by accessing the next logical block. In the new data structure the blocks are stored randomly, so it is necessary to 'link' those block which contain two parts of a line. A 'link' field is reserved in each text block for this purpose. Obviously, most of the text blocks will be linked together.

4.4 THE IMPLEMENTATION OF THE MAIN COMMANDS

In this section the implementation of the access, insert, and delete commands are described. Before this it is necessary to explain some of the variables and terms used in this section. The following figure shows the pointer block directory with some variables indicating its current status:

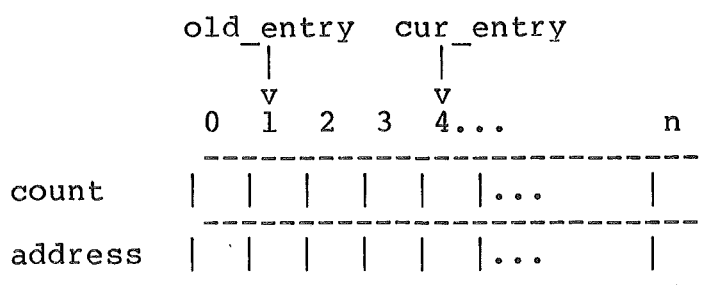


Figure 4.2

Pointer blocks Directory

'count' and 'address' are in fact two vectors making up the directory. They contain the number of pointers and the address of the pointer blocks respectively. 'cur_entry' and 'old_entry' indicate two entries of the directory corresponding to the blocks currently in the 'cur_window' and 'old_window' buffers respectively.

4.4.1 Accessing a Line

Access to a line is always done relative to the last line accessed (starting with the last work file line). The pointer to the last line accessed is in the pointer block corresponding to 'cur_window'. It is specified by two values: 'phys_cur_line' (physical current line) indicating its line number, and 'cur_offset' containing the offset of the line pointer within the current window. The following

figure shows the current window with its status information:

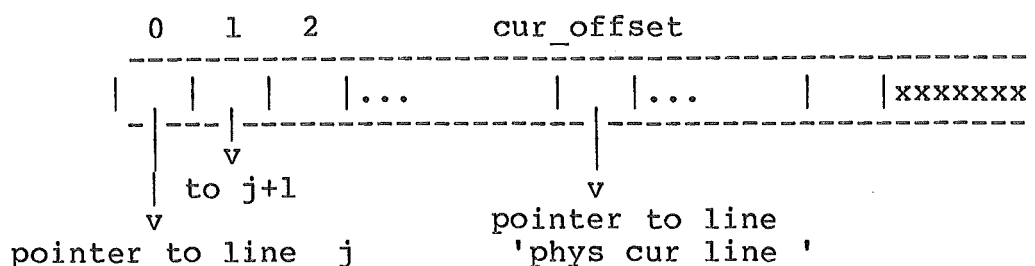


Figure 4.3

Current window with its current information

The pointer block containing the desired line pointer is accessed directly using the directory and the status information of the physical current line. In general, the access to line 'i' proceeds as follows:

1. It is determined whether the pointer to line 'i' is in the current window. It is in this window if:

$$(\text{phys_cur_line} - \text{cur_offset}) \leq i \leq (\text{phys_cur_line} - \text{cur_offset} + \text{count} ! \text{cur_entry})$$

('!' is the notation used in BCPL for array indexing)

2. If the pointer is in the current window, it is specified by the new values of 'cur_offset' and 'phys_cur_line' calculated as follows:

$$\begin{aligned} \text{cur_offset} &:= \text{cur_offset} - (\text{phys_cur_line} - i) \\ \text{phys_cur_line} &:= i \end{aligned}$$

3. If the pointer to line 'i' is not in the current window, it must be in a pointer block before the block specified by 'cur_window' (if $i < \text{phys_cur_line}$), or after it (if $i > \text{phys_cur_line}$). By traversing backward or forward within the entries of the

directory, the address of the pointer block containing the desired pointer is found and it is read into the 'cur_window' buffer, replacing the old one. In the end, the 'phys_cur_line' and 'cur_offset' are set to their new values.

4. The line pointer found in steps 2 or 3 above gives two values, as mentioned: The address of the text block (or an entry of the map_table), and the offset of the line.
5. The specified text block is read into the input_buffer, if it is not already there, and the text for line 'i' is copied into the buffer 'line'.

Since with this structure the pointer and text blocks are accessed directly, the same number of disk i/o operations are used as with CHEF for the access command. So the figures calculated in table III (chapter 3) are also correct in this case. It should be mentioned that there may be some extra computational overhead for the access in this structure as compared to CHEF. This is due to the calculations performed on the directory for traversing the entries. But, as it was pointed out before, the size of the directory is very small even for large files and this overhead is not significant in most cases.

4.4.2 Insertion

With this structure, the insertion of line pointers in the pointer-place does not affect any of the pointer blocks after the point of insertion, and it is completely local to the pointer block(s) involved. This results from the use of

the directory, and also from keeping some free space at the end of each pointer block. there are two mechanisms for insertion, depending on where the new lines come from:

a. If the new lines come from the terminal or another file, the insertion proceeds as follows:

1. The pointer block where the insertion is to be done is accessed and read into the current window. This is done by accessing the line at the point of insertion.
2. This block is divided into two from the point of insertion, and the right part is moved to the buffer referenced by 'old_window'.
3. The new line pointers for the inserted text are placed in the cur_window after the point where it was divided, without affecting any other part of the structure. If the current window is filled, it is written out on disk, and a new pointer block entry is created by expanding the directory. In this operation, the text for the new lines is stored in the last block of the record-place (currently in the buffer referenced by 'output_buffer').
4. After the insertion is completed, the pointers in the buffers corresponding to 'cur_window' and 'old_window' are either concatenated into one, if their total is less than the maximum allowable in one block, or otherwise distributed equally between the two blocks. In the latter case, a new pointer block entry is created in the directory.

b. If the new line comes from another region of the

work file, the insertion is done as follows:

1. The source and destination blocks are read into the buffers 'cur_window' and 'old_window', respectively.
2. The old window is divided into two from the point of insertion. The right part is stored in a newly allocated block.
3. The required pointers are shifted from the 'cur_window' buffer to the 'old_window' buffer. In this case there are two possibilities:
 - i. If all the pointers in the current window are shifted into the old window, and there are still more pointers to be inserted, the next source pointer block is read into the 'cur_window' buffer. In this case, if the modifier of the command is 'D' (delete), the first source block is deleted.
 - ii. If the total number of pointers in the old_window exceed the average, the buffer is written out on disk and a new block is created for more insertions.

The following figure shows an example of inserting 3 new lines from the terminal. Pointers to these lines, 4a, 4b, 4c are to be inserted after the line pointer 4. A maximum capacity of 10 pointers is assumed for each block. Figure 4.4(a) shows the changes in the 'cur_window' and 'old_window' buffers after each insertion step (described in 'a' above). Figure 4.4(b) shows the changes in the directory and the pointer blocks before and after insertion:

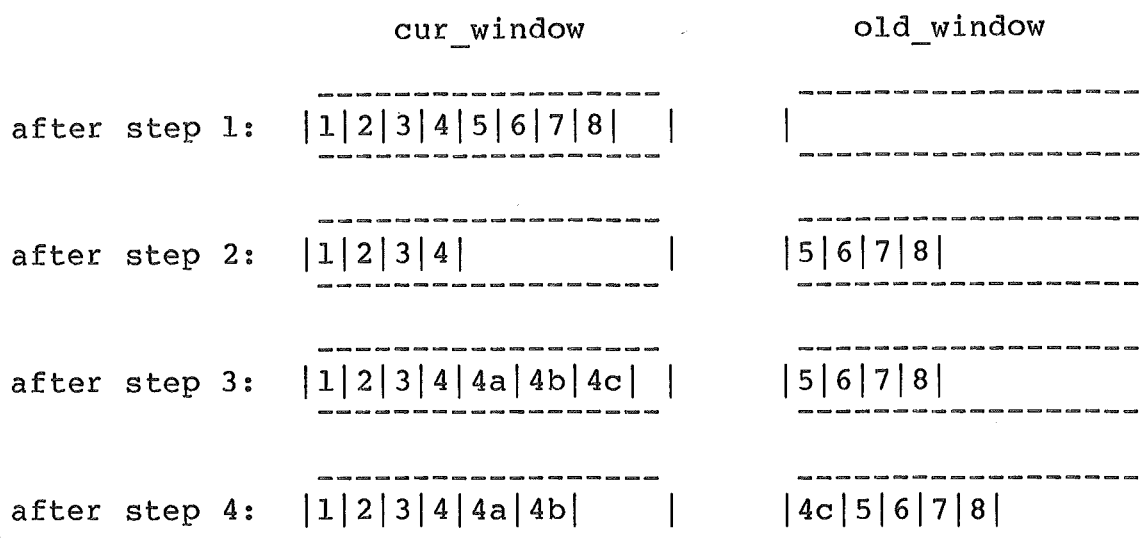


Figure 4.4 (a)

Changes in current and old windows for insertion

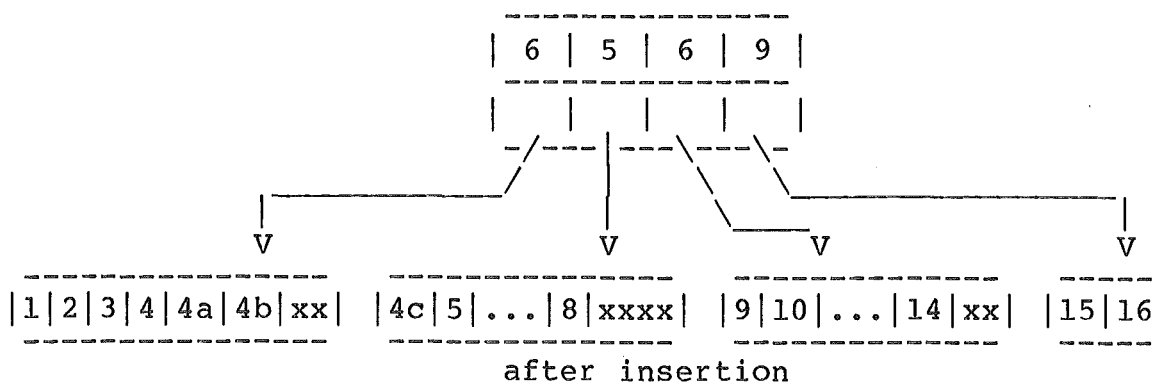
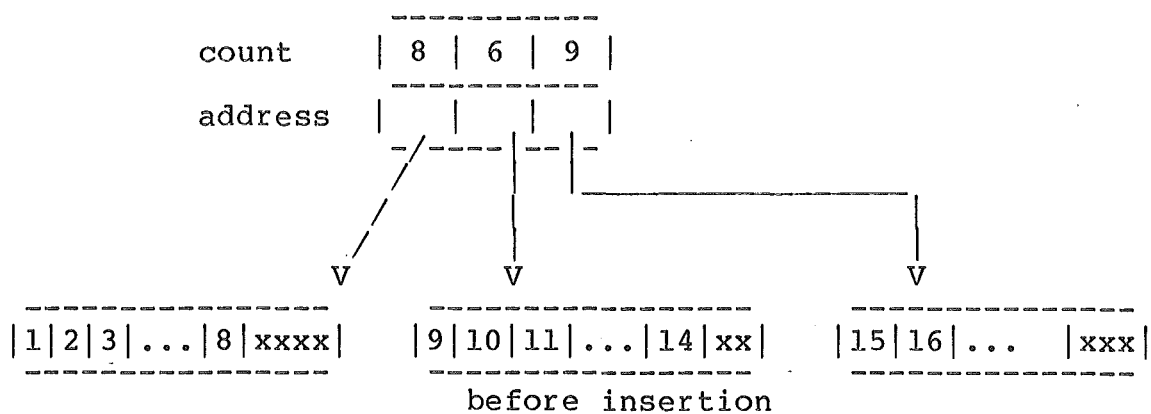


Figure 4.4 (b)

Directory and pointer blocks before and afetr
insertion

As can be seen, for a command such as 'ID' (insert from a work file region, and delete the source lines), deleting of the source line pointers is done at the same time as the insertion is performed. This was done in CHEF as a separate operation. We note that for short insertions, the only i/o operation which may be involved is when the line at the point of insertion is accessed. This is more efficient than a similar insertion in CHEF.

4.4.3 Deletion

The deletion of a line or a group of lines is also a local operation affecting only the block(s) where the pointers to be deleted are stored. The deletion is done in the following steps:

1. If all the pointers to be deleted are stored in a single pointer block, the deletion is simply done by shifting the pointers that follow the deleted region. In this case the corresponding 'count' is updated.
2. If the line pointers to be deleted are stored in two or more blocks, the following is done:
 - i. In the first block, all the pointers following the first pointer to be deleted are simply deleted by updating the block count in the directory.
 - ii. The next pointer block is read in, and if all the pointers in this block are in the range to be deleted, the whole block is deleted by deleting its entry from the directory.

This step is repeated as long as all the pointers in the successive blocks are in the range to be deleted.

- iii. For the last block, for which only a portion of its pointers must be deleted from the beginning, the deletion is done by shifting the pointers towards the beginning of the block.
3. At the end of the deletion operation, a procedure is called to redistribute the pointers between blocks if any block has less than the minimum number of pointers as a result of the deletion.

The following figure shows the directory and the pointer blocks before and after deleting lines 7 to 10 of a file:

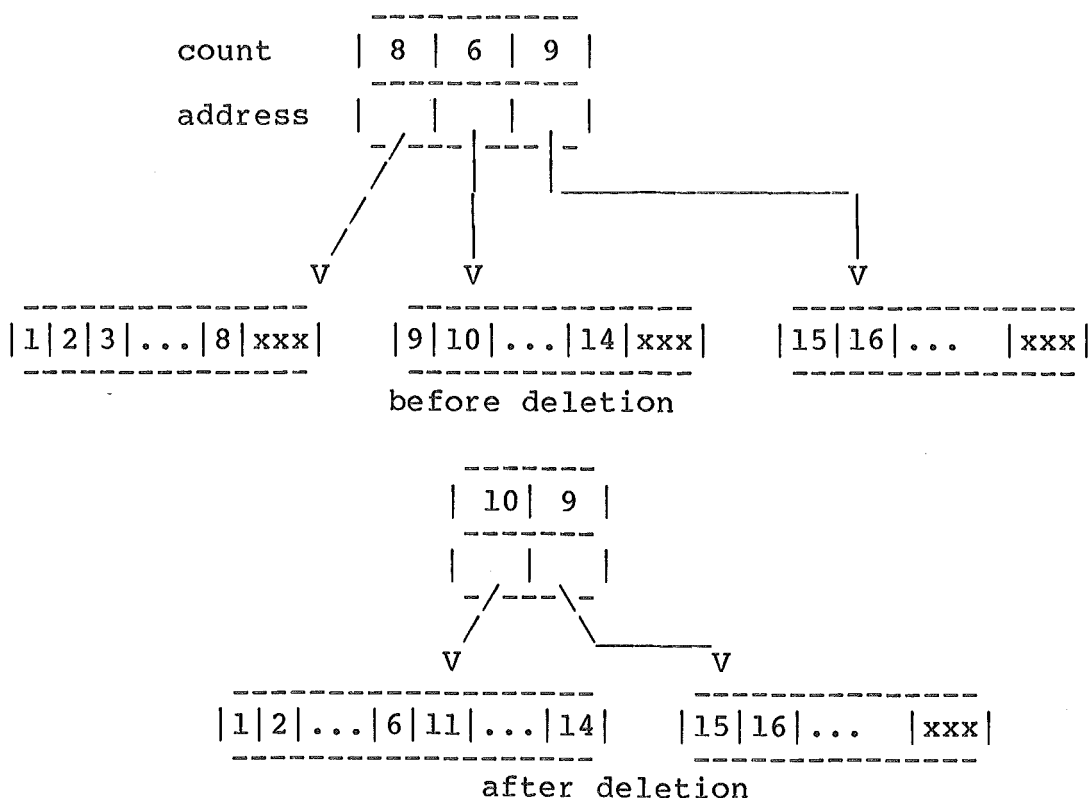


Figure 4.5

Directory and pointer blocks before and after deletion

It can be seen that, with the new data structure, the insert and delete commands are extremely efficient in terms of the number of i/o calls and the amount of pointer movement. These operations are also, to a great extent, independent of the file size being edited and the location of the operation within the file. In most of the data structures studied earlier, the efficiency of the main commands was directly dependent on the file size and the location of the operation. The relationship between the file size and the efficiency of the main commands will be shown for the new data structure by empirical results in chapter 6.

CHAPTER 5

EXTENDED FEATURES OF THE NEW DATA STRUCTURE

In this chapter, four important features of the data structure are described. This includes a technique of backup for the Undo command, the stacking of blocks, a garbage collection facility, and a recovery technique. The last three are new features of this data structure. The more efficient backup technique and the garbage collection were possible because of the inherent structure of the new design.

5.1 BACKUP FOR THE UNDO COMMAND

In this structure, a very similar method is used for the Undo command as in CHEF: The inverse of the commands that only move or copy the line pointers is computed and saved. For the commands that destroy the line pointers, copies of the pointers are stored in the backup-store. The main difference between the two data structures is in the structure of the backup blocks. As it was described before, in CHEF there is an associated backup block for each pointer block, reserved for the possible backup of pointers. At any time, only a (small) number of these blocks are used, i.e. the blocks corresponding to the pointer blocks where the operation was performed. With a small change in a pointer block (even a single pointer), the whole block is stored in the corresponding backup block.

In the new data structure, all blocks are allocated 'randomly', so it is possible to maintain the backup blocks independently of the other types, with a different structure. It was decided that a linked list of backup blocks is an efficient structure for this purpose, since these blocks are always accessed and manipulated sequentially. There is a buffer in memory which always contains the last backup block. The pointers which are deleted or altered are stored in this buffer, until it is filled, in which case it is written out on disk, and is freed for storing more pointers. To maintain the list structure, before writing the buffer out on disk a new block is allocated for the next backup block, and its address is stored on the 'link' field of the buffer. The first backup block (on disk) is always addressed by a variable called 'backup_header'.

The following figure shows this structure:

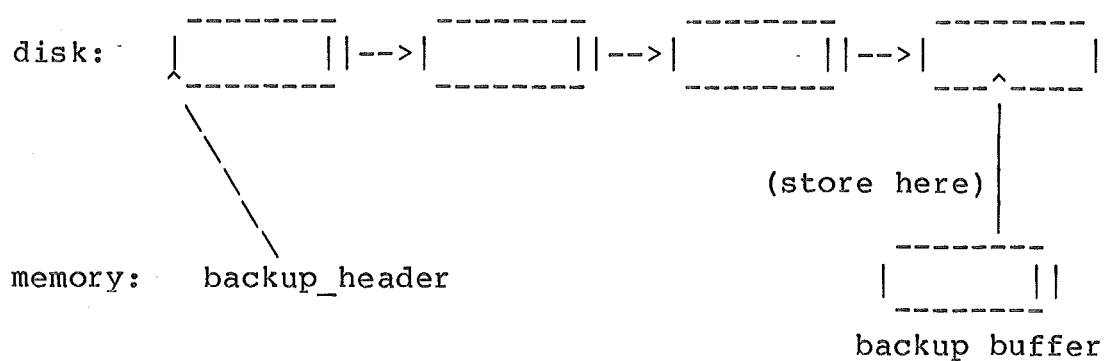


Figure 5.1

Structure of backup blocks for undo command

After all the pointers are stored in the backup-store, they remain there until one of the following conditions occurs:

1. The Undo command ('U') is encountered. In this case, the pointers must be inserted or shifted back to the pointer-place. As in CHEF, if the modifier of the 'B' operator is 'I' (insert), the pointers are inserted in the old positions from where they were deleted. If there is no modifier for the 'B' operator, the pointers are copied to the pointer-place, overwriting the existing ones. Backup-store is then reinitialized for storing new pointers in future.
2. a 'willing' or 'unwilling' command is encountered. In this case, a garbage collection is performed on the text blocks (described later in this chapter), by accessing them through the pointers of the backup-store. At the end of the operation the pointers are either replaced by the new pointers (if it is a willing command) or ignored (if it is an unwilling command).

Access to the backup blocks is made by the following steps:

1. The 'link' field of the backup buffer is set to zero, and it is stored on disk.
2. The block addressed by the 'backup_header' is read into the buffer.
3. After the pointers in the buffer are accessed, the next block is read in using the 'link' field of the block in the buffer. The last block is recognized by a 'link' field of zero.

After each block is accessed, it is either deallocated by the disk manager (in a dedicated disk), or

pushed onto a 'stack' for future use (in a file system).

There are several advantages gained by this structure:

1. There are no i/o calls for small backups (less than a block size). This will improve the general performance, since the backup procedure is executed very often (after every 'willing' command).
2. There is more efficiency in space, since no permanent file or disk blocks are allocated for the backup (as is done in CHEF), and the blocks are allocated and deallocated dynamically as required.
3. The 'control lines' can also be undone, by storing their 'contents' in the backup_buffer before changing them.
4. It assists the implementation of garbage collection, explained later.

5.2 STACKING OF UNUSED BLOCKS

Stacking is the facility used to allow reallocation of unused blocks or map-table entries.

Stacking is done for two cases:

1. When the data structure is implemented on top of a file system.
2. When a map-table is used as part of the structure.

It is clear that in this structure, all three types of blocks used are dynamically allocated and deallocated, i.e. pointer blocks as a result of delete and insert commands, backup blocks as explained above, and text blocks as a

result of garbage collection. If the deallocated blocks are simply ignored and are not used again, the address space of a text pointer will soon be exhausted, and hence no more editing will be possible. This problem is solved as follows: In the first case (a file system), all three types of text, pointer, and backup blocks are pushed onto the stack after being collected as garbage (text), deleted (pointer), or released (backup).

In the second case (map-table), only the map-table entries which are collected as garbage are pushed onto the stack. Here the deallocation of the physical text, pointer, and backup blocks is done by the disk manager (e.g. by using a bit map).

There is a boolean variable called 'stack_empty', which shows whether there is any entry on the stack. When a new block is required, the stack is checked first and, if there is any entry in it, it is allocated as the new block, otherwise the next logical file block or map-table entry is allocated.

If the maximum file size is 256 blocks, a single buffer of 256 words will be enough for the stack. If it is larger than 256 blocks, it is possible to maintain a 'linked list of stack blocks'. The top portion of the stack is always in the 'stack_buffer', and a 'link' field in this buffer points to the previous portions on disk. The variable 'stack_empty' or the 'link' field of the buffer can be used to determine whether there are more stack blocks on disk.

This linked list mechanism for the stack blocks has not been implemented in the present version of the editor and a

single memory resident buffer (with its length to be set by the implementor) is used for the stack.

5.3 GARBAGE COLLECTION

In the CHEF data structure when a line is deleted or altered, its actual text is simply ignored (abandoned), by deleting or replacing its pointer. The new or altered line text is stored at the end of the record-place. In this way the record-place is continually growing, and hence approaching the maximum that the line pointer can address. At the same time there may be several 'garbage' areas along the record-place which could be used for storing the new or altered lines.

In the new data structure, garbage text blocks are returned to the system for future use as described below.

There is a (line) 'reference_count' field within each text block, which holds the number of lines stored on that block. When the lines are first stored in the block, the reference count is incremented for each new line. For the lines which continue into the next block, the reference counts of both blocks are incremented. Later during the editing, if a line within the block is deleted, the reference count is decremented. In this case, if the reference count becomes zero, the block number or the map-table entry is pushed onto the stack. In the case of a dedicated disk, the physical block is also deallocated.

It was explained before that for deleting or replacing a line, only its pointer is deleted or replaced. This pointer is used to access the actual line for garbage

collection. If the line is collected immediately after the command is executed, it will not be available for a possible 'U' (undo) command. So there should be a mechanism to postpone the garbage collection temporarily to make sure that the next command is not 'U'. This task becomes easy if we use the backup-store pointers, since these pointers are most of the time the ones that were deleted or altered by the last command. So garbage collection is only performed whenever the backup pointers are not required anymore, i.e. whenever there is a 'willing' or 'unwilling' command. The garbage collection is done by reading the line texts (blocks) accessed by these pointers, and decrementing the reference counts, as explained above.

In this way, we have effectively postponed the garbage collection one stage, but there are some problems which stem from using the backup-store for garbage collection. These problems and the solutions are described below.

1. If several new lines are stored in the work file ('I' operator) and the next command is 'U', the pointers to these lines will be deleted in the pointer-place, but their text will not be collected as garbage. This problem is solved by storing the pointers in the backup-store whenever the editor is in the 'undo-state' and the operator is 'D' (the inverse of 'I').
2. In the CHEF editor, if the operator is 'R' (replace), all the lines to be searched for the pattern are stored in the backup-store. With the next 'willing' or 'unwilling' command, a garbage collection will be performed on these lines. In this way, some line texts

may be lost, since the 'R' command replaces only those lines that match the search pattern. to solve this problem, only the line pointers that are replaced are stored in the backup-store. These pointers have to be stored together with their line numbers, in order to restore them in the proper places if the next command is 'U'.

3. The 'X' (execute) and 'XF' (execute from a file) cannot be undone in CHEF. For this reason if these commands delete or alter line pointers (by executing a 'willing' command), the old line texts are not stored in the backup-store, and hence they are not collected as garbage. To collect these lines, their pointers are stored in the backup-store. For the 'X' command, they are stored together with their line numbers (same format as the 'R' command), and for the 'XF' command they are stored as normal.

The garbage collection stacks for future use the file block number or the map-table entry of released blocks, so it has the advantages given in the last section for stacking. It is realized that garbage collection has some overheads in terms of the number of i/o calls, due to the access to the actual text, but this is easily justified by the advantages gained. The effect of garbage collection on the performance of the editor is measured in chapter 6.

5.4 RECOVERY FROM SYSTEM FAILURE

In the old editor, if during an editing session a hardware or software failure occurs (e.g. a power failure), the work file is not recoverable and all the editing efforts of that session are lost. A worse situation is when a failure occurs during the writing of the edited work file back to the original file (the 'WF.' command). In this case both the work file and the original file will be lost.

To solve this problem for the new data structure a recovery mechanism can be used as described below. At the beginning of the editing session, when the command 'EF' is entered, the editor reads and stores a copy of the file in 'work_file_1'. Then it stores the initial values of all the necessary working variables and tables in specified places in 'work_file_1', and dumps a copy of this file onto the 'work_file_2'. Now the editor starts normal operation with 'work_file_1'. After the first alteration (a willing or unwilling command), 'work_file_1' will be in an 'inconsistent' state, while 'work_file_2' is in a previous 'consistent' state. These states are indicated by a flag in each work file (called 'consistent_state_'). If a failure occurs while the editor is working on 'work_file_1', it is possible to recover to the previous stage by resuming the operations on 'work_file_2'.

After a certain number of operations on one work file (e.g. 'work_file_1'), the editor switches to the other file ('work_file_2') by performing the following steps:

1. All the working variables and tables are stored in 'work_file_1'.

2. The 'consistent_state_' of 'work_file_1' is turned to TRUE.
3. The 'consistent_state_' of 'work_file_2' is turned to FALSE.
4. The contents of 'work_file_1' are copied onto the 'work_file_2'.

Now the editor can continue operations using 'work_file_2'. If the failure occurs during copying the 'work_file_1' onto 'work_file_2' (step 4), 'work_file_1' is in a consistent state and can be recovered.

When the system starts after a failure, the recovery procedure is called (either automatically or by a new command such as 'EW') to restore the work file as follows:

1. It checks the 'consistent_state_' flags of both work files and finds the one with TRUE state.
2. It reads all the recovery variables and tables from the file into the corresponding places in the editor's memory space.

The period after which the editor switches to the next file is determined by the number of editing operations (alterations) performed on the work file, rather than the actual time period. The number of alterations can effectively be obtained by counting the number of willing and unwilling commands. This (fixed) number can be set by the implementor and is a trade-off between the speed of the editor and the importance of the files being edited.

The necessary variables and tables which are stored for recovery purposes are as follows:

1. The Directory
2. The map-table (if any)
3. The pointers to control lines
4. All the working variables such as 'last_line', 'next_blk_n', 'last_blk_n', etc.

These variables and tables are stored in randomly allocated disk or file blocks, except for the first block which is stored in a fixed block. In a dedicated disk environment, the first block is pointed to by a pointer in the 'disk directory' (there is one entry for each active user in the disk directory). The recovery blocks are 'linked' to each other in the same way as backup blocks and stack blocks.

To create two work files is a simple task when a file system is used. With a dedicated disk this task becomes more complicated and the disk manager must handle it. This can be done by keeping one pointer to the first block of each work file (the recovery block) in the disk directory.

The recovery feature has not been implemented in the present version of the editor.

CHAPTER 6

IMPLEMENTATION OF THE NEW DATA STRUCTURE AND THE RESULTS

This chapter is divided into two sections: In the first section the basic implementation of the new data structure is described. This includes a general description of the new, modified, and deleted sections of codes. In the second section some tabular comparisons and measurements, which were produced by running both the old and new editors in sample sessions, are presented.

6.1 THE IMPLEMENTATION OF THE NEW DATA STRUCTURE

The new data structure was implemented in a version of CHEF on the ECLIPSE S/130. This section gives a general description of all the new procedures produced and the old procedures affected in the implementation. It is not meant to be a detailed study of these procedures. The interested reader can refer to the procedure listings in the appendix.

The CHEF program is written in the BCPL Language. It consists of 11 separate modules(sections), named EF0 to EF10. EF0 is the 'header' section, which contains all the global variable and constant declarations, and is used by other sections for separate compilations. The program sections are designed and organized in a structured manner, which makes the modifications and enhancements very easy. Section EF6 contains all the procedures which are concerned with the manipulation of the work space. The main

if it was changed since last read in.

fetch grab(i): Fetches the grab (pointer) of line 'i'.

fetch record(grb): Fetches the line pointed by 'grb' into the buffer 'line'.

grab ad(i, store): Yields the address of a word (in the buffer 'cur_window') containing the pointer to line 'i'. If 'store_' is true, the window is being altered by the calling procedure. This is called by several procedures.

store grab(i, g): Stores the grab 'g' of the 'i'th line in the pointer-place.

store new record(): Stores the contents of 'line' at the end of the record_place and yields the grab position at which it was stored.

store record(grb): Stores the contents of 'line' in the record_place at the grab position 'grb'.

set up control(): Allocates the maximum possible space for each of the control lines, and initializes their contents.

There are another three procedures in section EF4 which call one or more of the above procedures to carry out their functions:

delete lines(l1, l2, r): Deletes 'r' lines of the work space from 'l1' to 'l2' by calling the procedure 'copy_lines()'.

insert lines(dot stop): Inserts lines from the terminal, if 'dot_stop_' is true, or from a file or other region of the work space, if 'dot_stop_' is false, after the line 'l_line1'.

expand(index, size): Expands the array of line

pointers at position 'index' by 'size' cells.

For implementing the new data structure in CHEF, most of the changes were made in section EF6. There were also some changes in sections EF8 (undo command), EF4 (command interpretation), EF1 (initial memory allocation), and EF0 (header). The following paragraphs are a general description of these changes.

Accessing and storing the lines: The main procedures in EF6 for access and storage of lines are modified versions of 'fetch_line()' and 'store_line()'. The procedures 'fetch_grab()', 'store_grab()', and 'grab_ad()' are replaced with new procedures 'fetch_pointer()', 'store_pointer()', and 'pointer_ad()' respectively. The procedure 'pointer_ad(i, store_)' uses the directory to find and read in the pointer block containing the pointer to line 'i'. The procedure 'store_new_record()' is deleted and a new procedure 'store_block()' is added to store the text block on disk or file.

Insertion: For insertion the procedure 'do_i()' in EF4 is modified. The procedures 'insert_lines()' in EF4 and 'make_space()' and 'move_windows()' in EF6 are deleted, and four new procedures 'make_space()', 'insert_new_lines()', 'insert_existing_lines()', and 'distribute_space()' carry out the insertion as follows: 'make_space()' is called to divide the pointer block containing the 'cur_line' into two. Then one of the procedures 'insert_new_lines(dot_stop_)' (if the lines comes from the terminal or a file), or 'insert_existing_lines(l_off)' (if the lines come from another region of the present work file) are called to do

the insertion. The procedure 'distribute_space(old_entry_)', which is also used when deleting the lines, is called at the end of insertion to distribute the pointers equally between the adjacent blocks, or concatenate them, for any block containing less than the minimum number of pointers.

Deletion: The procedures 'delete_lines()' in EF4 and 'copy_lines()' in EF6 are deleted, and the new delete operation is performed by the procedure 'delete_lines()' in EF6. At the end of the deletion, 'distribute_space()' is called as above.

Directory manipulation: Traversing the directory entries for finding the required pointer block is done in the procedure 'pointer_ad()'. Adding a new entry to the directory (for assigning a new pointer block) is performed by the new procedure 'add_entry(entry)'. The deletion of the directory entries (for deleting the pointer block) is done by the new procedure 'delete_entry(entry)'.

Allocation and deallocation of blocks: Two procedures 'allocate_a_blk()' and 'next_blk()' are used for allocating the blocks. 'next_blk()' is only used when there is an underlying file system or the map_table. 'allocate_a_blk()' calls 'allocate_phys_blk()' when used in a dedicated disk environment. In the present implementation of the editor (on the ECLIPSE S/130) 'allocate_phys_blk()' has only to allocate sequential blocks from the file. When there is no underlying file system, the actual procedure for allocating the disk blocks can be implemented by using, for example, a bit map. Deallocation of the blocks is done by storing them

in the stack (with a file system or map_table). The deallocation of the physical disk blocks is done by 'deallocate_phys_blk()' for a dedicated disk. In the S/130 implementation this procedure has nothing to do.

Stacking: Stack manipulation is done by the procedures 'push_stack(blk_entry)', and 'pop_stack()' , for storing and retrieving the blocks respectively. The variable 'stack_top' always indicates the last stack entry.

The environment specification: There are two manifest constants (in EF0) called 'dedicated_disk_' (boolean) and 'computer_bits_no' (integer, for keeping the word size of the computer). which are set by the implementor to specify the existing environment. Using conditional compilation, the BCPL compiler will produce different object code for the editor in different environments. In particular, when the constant 'dedicated_disk_' is set to true and 'computer_bits_no' to 16 (a 16-bit computer is used), the map-table is not created, and therefore all sections of code which are concerned with the manipulation of the map-table are eliminated from the object code.

Garbage collection: The garbage line texts are returned to the system by the following three procedures: 'collect_pointers()', which accesses all the garbage pointers in the backup_store, 'collect_text(lr)', which accesses the text blocks, and 'collect_garbage(buffer, blk)', which decrements the reference count of the text block in 'buffer', and stacks the block if the reference count is zero.

Backup procedures: There are two main procedures for

the manipulation of the backup blocks: 'store_backup(l1, l2)', which stores the pointers 'l1' to 'l2' in the backup blocks, maintaining their linked structure, and 'fetch_backup(l1, l2, insert_, off)', which uses one of the local procedures 'insert_backup(lr, off)' (if the modifier of 'B' operator is 'I') or 'shift_backup(lr, off)' (if there is no modifier) to insert or copy the backup pointers back to the pointer_place for undoing a command.

Control lines: The procedure 'set_up_controls()', for initializing the control lines, is modified. The manipulation of the control lines is done by a single procedure 'fetch_store_control(i, store_)'. If 'store_' is true, it stores the buffer 'line' in the control line 'i', otherwise it fetches the contents of the control line 'i' into the buffer 'line'.

Initialization of the workspace: In the old CHEF, the procedure 'init_work_space()' (in EF1) was used to initialize a fresh work space and possibly to read in a file. This procedure is modified to initialize other variables and tables such as the directory and the map-table. This procedure was transferred to section EF6 because of its dependence on the data structure.

The rest of the procedures in EF6 which are left unchanged are as follows: 'clear_all_flags()', 'flag_the_line()', and 'was_flagged()', used by the 'X' command, 'do_z()', used for implementation of the 'Z' command, 'empty_work_space()', used by the 'E' command, and 'stack_work_space()', and 'unstack_work_space()', used by the 'N' and 'Q' commands for stacking and unstacking of the

work space.

The following procedures in section EF8 were modified for undoing a command:

do b(): calls the procedure 'fetch_backup()' to retrieve the backup pointers. If the modifier is 'I' it calls 'make_space()' and 'distribute_space()' before and after fetching the pointers.

post trail(): This procedure is used for computing the inverse of the editor commands for undoing. In the new data structure, it is enhanced to compute also the inverse of the commands that change the control lines.

A listing of the new EF6 section is included in the appendix.

6.2 EXPERIMENTAL RESULTS AND COMPARISONS

As mentioned, the new data structure with the above features was implemented on a version of CHEF on the ECLIPSE Computer. Software monitors were installed in both the original and new versions of CHEF to record some measurements for comparisons. As was expected, the number of block i/o calls was the major factor in the response time of commands. For each command five values were recorded: The number of calls to read pointer blocks, the number of calls to write pointer blocks, the number of calls to read text blocks, the number of calls to write text blocks, and the response time of the command. For obtaining the response time, the ECLIPSE real-time clock was used in a single user environment, so that it is equivalent to the

execution time of a command. All response times are in units of 10th of a second. It must be mentioned that a block read or write recorded by the monitor is in fact an i/o call to the file system and does not necessarily correspond to one physical disk transfer. Only the total number of i/o calls are shown in the following tables (except for table XI).

For these measurements typical commands were executed to show the main differences between the two data structures. These commands are as follows:

1. Commands for the basic operations access, insert, and delete.
2. Commands for the undo operation to show the difference between the new and old backup implementations.
3. Commands for showing the effect of garbage collection and evaluating the performance of the editor with and without garbage collection.

(1) Commands for access, insert, and delete

There is no major difference for access operations. From the commands for insert and delete we can draw the following conclusions:

- i. There is a general improvement in the performance of the CHEF editor with the new data structure.
- ii. The performance of the editor with the new data structure is not dependent on the file size being edited.
- iii. The performance of the editor with the new data structure is not dependent on the location of the operation within the file.

For demonstrating the second and third points, similar commands were repeated in different locations of different sized files. The locations are typically the beginning, the middle, and the end of the workfile. Two sample files with 1000 lines (file called t1000) and 3000 lines (file called t3000) are used for the experiments. It is important to mention that the values given in these tables for the number of i/o calls are only correct for the cases where there is no garbage line (as a result of a previous delete and replace operation) to be collected after the execution of the corresponding command. If there is any garbage line the number of i/o calls will be higher because of the extra accesses to text blocks for updating the reference counts. The overhead associated with the garbage collection will be evaluated later in this section.

Access Commands: Common operators for accessing the workfile lines are the 'P' (print) and 'V' (view) operators. Also two operators 'R' (replace) and 'T' (tag) are included in the following table as access commands since they are implemented as a number of access and (possibly) store operations.

command description	total i/o calls		response time	
	old	new	old	new
enter file 't1000' into workspace	13	14	34	34
display last line	0	0	0.5	0.5
display line 500	0	0	0.5	0.5
display first line	0	0	0.5	0.5
replace all occurrences of 'xx' by 'yy'	19	23	23	25
tag all lines with 'w'	34	38	28	31

Table V

Results for typical access commands
in a 1000-line file

command description	total i/o calls		response time	
	old	new	old	new
enter file 't3000' into workspace	45	45	92	92
display last line	0	0	0.5	0.5
display line 2000	0	0	0.5	0.5
display line 1000	0	0	0.5	0.5
display first line	0	0	0.5	0.5
replace all occurrences of 'xx' by 'yy'	65	70	69	70
tag all lines with 'w'	110	116	89	89

Table VI

Results for typical access commands
in 3000-line file

As it is seen from these tables, there is no great difference between the two structures for commands that

access the workfile lines. This was expected from the design of the new data structure as discussed in chapter 4.

Insertion: The insertion of new lines can be from the terminal, an external file, or another region of the workfile. It is not possible to record a realistic response time for commands which do insertion from the terminal because the user's typing time is also included. But this type of insertion is similar to insertion from an external file which is presented in table VII. Tables VIII and IX show the results of typical commands for insertion from another region of the workfile.

command description	total i/o calls		response time	
	old	new	old	new
enter file 't1000'	13	14	34	34
insert file 't10' after last line	0	0	4	4
insert file 't10' after middle line	12	0	12.5	4
insert file 't10' after first line	15	0	17.5	4
enter file 't3000'	45	45	92	91
insert file 't10' after last line	1	0	4.5	4
insert file 't10' after line 2000	20	0	16.5	4.5
insert file 't10' after line 1000	36	0	25	4.5
insert file 't10' after first line	47	0	34	4.5

Table VII

Results for inserting an external file in a
1000-line and 3000-line file

command description	total i/o calls		response time	
	old	new	old	new
enter file 't1000'	13	14	35	37
insert last line before first line	7	5	6.5	2
insert last line after middle line	6	5	4	2
insert first line after last line	0	0	0.5	0.5
insert lines 200-300 after last line	8	6	4	2.5
insert lines 700-800 before first line	10	6	9	2.5
insert all lines before first line	24	15	20	6
insert after last line and delete lines 1-10	8	8	7.5	3
insert before first line and delete lines 900-1000	13	7	10	3.5

Table VIII

Results for inserting from another region of
the workfile for 1000-line file

command description	total i/o calls		response time	
	old	new	old	new
enter file 't3000'	45	45	93	93
insert last line before first line	24	6	20	2.5
insert last line after line 1000	19	6	13	2.5
insert first line after line 2000	10	6	8	3
insert first line after last line	1	0	0.5	0.5
insert lines 200-300 after last line	9	6	3.5	3
insert all lines before first line	73	32	56	12.5
insert after last line and delete lines 1-10	24	8	20	3
insert before first line and delete lines 2600-2900	45	11	33	3.5

Table IX

Results for inserting from another region of
the workfile for a 3000-line file

Deletion: Tables X and XI show the results of deleting a single line or a group of lines from different locations of two different sized files. In table XI, the number of i/o calls are recorded in detail to show the effect of the operation (delete) on the number of pointer and text block calls.

command description	total i/o calls		response time	
	old	new	old	new
enter file 't1000'	13	14	35	37
delete first line	10	0	7.5	0.5
delete middle line	9	0	5	0.5
delete last line	1	0	1.5	0.5
delete lines 100-500	12	6	7.5	2.5
delete lines 500-1000	7	6	2	2

Table X

Results for delete commands in a
1000-line file

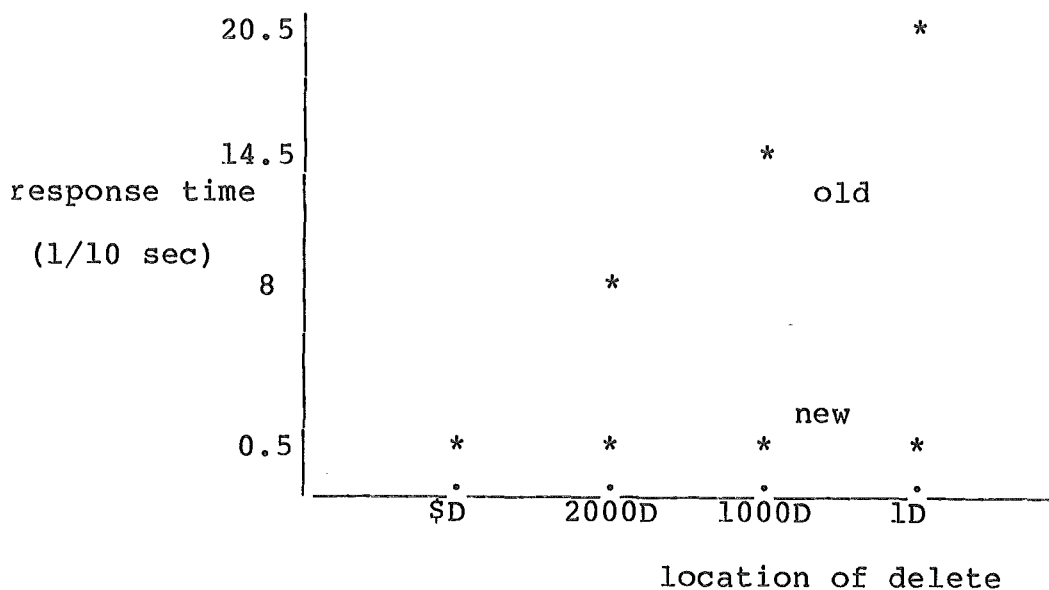
command	i/o calls								response time	
	old				new					
	pointer		text		pointer		text			
	in	out	in	out	in	out	in	out		
enter file 't3000'	0	12	0	33	0	12	0	33	93	90
delete first line	9	10	2	0	0	0	0	0	14.5	0.5
delete line 1000	5	6	2	0	0	0	0	0	8	0.5
delete line 2000	0	1	0	0	0	0	0	0	0.5	0.5
delete last line	21	13	2	0	6 (7)	4	0	0	22.5	4.5
delete lines 2000-2900	7	7	2	0	5	4	0	0	4.5	3.5

TABLE XI

Results for delete command in a 3000-line file
with detailed i/o

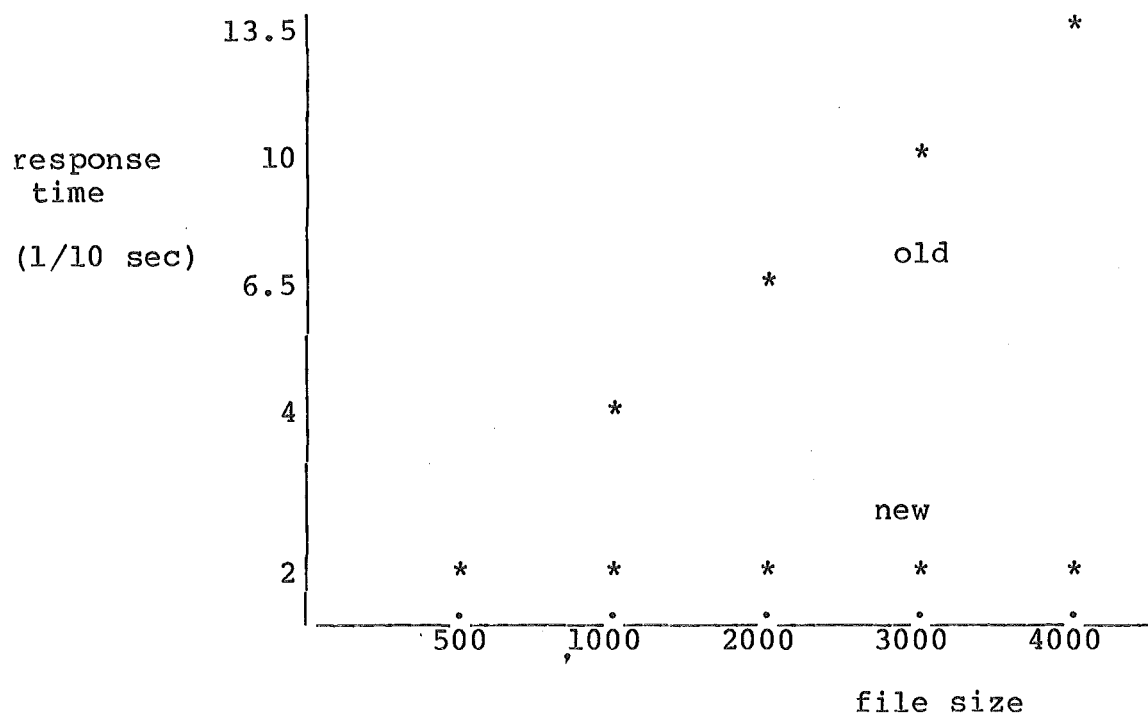
From the above tables for insert and delete commands it

can be seen that the number of i/o calls, and hence the response times, are reduced with the new data structure. The reduction is mainly in the number of pointer block calls which was expected as discussed in chapter 4. The following diagrams are graphical representations of some typical commands for insert and delete operations to show the relationships between the response time and file size/location of the operation. graph I shows the result of deleting a single line at different locations of a 3000-line file (from table XI). The second graph is the result of inserting the last line in the middle of different size files.



Graph I

Deleting a single line at different locations
of a 3000-line file



Graph II

Inserting the last line in the middle
of different sized files

(2) Commands for the undo operation

Tables XII and XIII below show the total i/o calls and response time of some typical commands and their reverse ('U' command) to compare the old and new backup techniques.

command description	total i/o calls		response time	
	old	new	old	new
enter file 'tl000'	13	14	34	34
delete first line	10	0	8.5	0.5
undo	8	0	8	0.5
insert last line after line 500 (middle line)	6	0	4	2
undo	6	0	4.5	0.5
delete last line	1	0	0.5	0.5
undo	2	0	1	0.5
insert lines 200-300 after last line	8	7	4	2.5
undo	1	0	1	0.5
delete lines 100-500	12	6	7.5	2.5
undo	13	10	7.5	4
insert file 'tl0' after line 500	12	0	12.5	4
undo	7	0	4.5	0.5

Table XII

Results for the undo command in a
1000-line file

command description	total i/o calls		response time	
	old	new	old	new
enter file 't3000'	45	45	92	93
delete first line	26	0	20.5	0.5
undo	24	0	19	0.5
insert last line after line 1500	14	6	10	2
undo	14	0	10.5	0.5
insert lines 200-300 after last line	8	6	3.5	2
undo	1	0	0.5	0.5
delete lines 100-900	36	10(11)	22.5	4.5
undo	36	11(14)	23	5
insert file 't10' after line 1500	29	0	21	4
undo	15	0	1.5	0.5

Table XIII

Results for the undo command in a
3000-line file

It can be seen from these tables that there is a general improvement in the response time of the undo command and it is not dependent on the location of the operation and the file size.

(3) Evaluation of garbage collection

To show some of the effects of garbage collection on the performance of the editor the following commands (table XIV) were executed in a typical session. The maximum pointer value is obtained by the CHEF command 'Z'.

command description	max pointer value		comment
	old	new	
enter file 'tl000'	10751	11601	
tag all lines with 'a'	20621	21933	
add 'x' to beginning of every line	30743	22407	see note (1)
tag all lines with 'b'	workspace overflow	22752	see note (2)
delete first line	32752	22752	
tag all lines with 'c'	workspace overflow	22835	
insert file 'tl0' after first line	" "	22854	
add 'y' to beginning of every line	" "	23330	
insert lines 200-300 after line 10	32761	23330	
tag all lines with 'd'	workspace overflow	24967	

Table XIV

Effect of garbage collection in
pointer values

notes:

- (1) The second command above (tag with 'a') replaces all lines of the workfile with tagged lines and stores the original line pointers (created by the first command) in the backup-store. With the new data structure, these pointers values are returned and reused by the third command.
- (2) with the old structure, only 214 lines are tagged with 'b' before the overflow occurs and the maximum pointer value in the workspace (line 214) is 32752.

After the third command in the above table the editor with the old data structure cannot carry out any operation which includes the replacement or insertion of new text, and

it replies with the error message 'workspace has overflowed'. This is because the pointer value has reached the maximum value that can be represented by a 16-bit computer (32767). But the editor with the new data structure can continue operations as usual. The reason, as described in chapter 5, is that the previous pointer values which are not used anymore are collected and returned to the editor dynamically.

The following tables show the overheads incurred by using the garbage collection. This overhead is mainly because of extra i/o calls for reading the text blocks (to update the reference counts). In this table the response times and number of i/o calls are recorded for the editor with the new data structure both with and without garbage collection. Similar commands are executed for two files with averages of 20 characters per line (file 't1000.20' in table XV) and 40 characters per line (file 't1000.40' in table XVI) to show the effect of the average line size on garbage collection.

command description	total i/o calls		response time	
	with GC	no GC	with GC	no GC
enter file 'tl000.20	50	50	110	110
add 'x' to beginning of lines 1-500	57	57	49	49
view line 1 ('V')	3	3	1	1
print last line('P')	2	2	1	1
delete line 1	29	0	9	1
delete line 1	0	0	1	1
tag lines 1-10	1	1	1	1
delete line 1	2	1	1	1
tag lines 1-100	12	11	7	7
delete line 1	10	1	4	1
tag lines 1-200	20	19	12	12
delete line 1	14	0	5	1

Table XV

Overheads of garbage collection in a
1000-line file

command description	total i/o calls		response time	
	with GC	no GC	with GC	no GC
enter file'tl000.40'	89	89	162	162
add 'x' to beginning of lines 1-500	97	97	84	83
view line 1 ('V')	3	3	1	1
print line 1000('P')	2	2	1	1
delete line 1	49	0	15	1
delete line 1	0	0	1	1
tag lines 1-10	3	2	1	2
delete line 1	3	0	2	1
tag lines 1-100	22	19	12	11
delete line 1	13	1	5	1
tag lines 1-200	37	36	22	19
delete line 1	23	1	7	1

Table XVI

Overheads of Garbage Collection in a
3000-line file

We can conclude from these tables:

1. The garbage collection mechanism does not affect the neutral commands (e.g. 'V', 'P').
2. The amount of the overhead with garbage collection is dependent on the number of lines affected by the last willing command. This can be realized by comparing the different response times (or i/o calls) for the commands which delete line 1 in each table.
3. The amount of overhead is also dependent on the average number of characters per line. This can be realized by comparing the two tables.

CHAPTER 7

CONCLUSION AND THE FUTURE7.1 CONCLUSION

This thesis has presented the design and implementation of a new data structure for the workfiles of multiuser line-oriented text editors. This editor is a logical improvement of the CHEF editor data structure which, on the one hand, increases the performance of the editor and on the other hand, should be suitable for a multi-user editing server in a local-area network environment. As was discussed in chapter 3, the CHEF data structure utilizes a sequential structure of disk (or file) blocks. But in a multi-user editing server it would be more efficient to dispense with the underlying file system, particularly if the number of users served by the editor is high. Therefore a main objective was to incorporate this capability within the editor. It was later realized that this objective would in turn help in achieving a second objective which is an increase in the performance of the editor by obtaining a faster response time for the basic editing commands. A study of the data structure of a number of other editors was presented in chapter 2. These data structures were divided into 6 categories on the basis of their historical development and performance. The advantages and disadvantages of each category were discussed and the main problems affecting the response times of the editing

commands were shown to be: the large number of disk i/o operations, the extensive movement of characters, and the need to reorganize the workfile after a certain amount of editing had taken place. It was seen that probably the most important development of editor data structures started with those utilizing an array of line pointers. In these editors the actual text remains unchanged and sequential and the main question becomes what structure to use for the line pointers (rather than the actual text) to solve the above problems. CHEF, described in chapter 3, was one of such editors in which the array was stored on disk and a simple virtual memory technique was used for its management. This solved the above problems to a great extent, but the number of disk i/o operations was still high for insert and delete commands.

In the design of the new data structure, presented in chapter 4, a more efficient structure is suggested for the array of line pointers by utilizing an internal directory for the pointer blocks. This allows fast response times for most editing commands because of a considerable reduction in the pointer block i/o operations and pointer movements. The reduction in the number of i/o calls is of particular interest since it is a major factor in the performance of editors which utilize disk storage for their workfiles. The large number of disk i/o operations in previous editors was due to the fact that the effective size which was manipulated by the editing commands (expansion, contraction, etc.) was the entire text. By using line pointers to the text the effective size reduces to the size of the pointer array. In the new data structure the effective size is

further reduced to the size of the pointer block directory (in memory) and/or the local portions of the array.

Another advantage of using an internal directory is that it allows for the use of random (disk) blocks for storing the pointers. This is a part of the first design objective which was to remove the dependence on an underlying file-management system.

Some of the extended features described in chapter 5 are also facilitated by the new design. The backup of pointers for the undo command is done local to the pointers involved and hence its efficiency is increased. The garbage text blocks and the unused pointer values are collected and returned to the editor. The concept of using random blocks also allows for storage of the necessary information in linked list of blocks. The linked list structure is used in 3 different cases: the backup of pointers for the undo command, the stack of released entries (from garbage collection, etc.), and the storage of working variables and tables for recovery purposes.

7.2 THE FUTURE

All the features described are implemented (unless otherwise noted) in the version of CHEF which incorporates the new data structure. In the future there are two areas in which the editor can be expanded and utilized:

1. As an editing server in the ring network of the Department of Computer Science. In this usage one copy of the editor can reside in the memory of a microprocessor (possibly a Motorola 68000), and it

serve the editing requests of all users of the network. For each user, the editor would get a copy of the original file from the file server and store it in random blocks of a dedicated disk. There would be the requirement for a simple disk space manager for functions such as the allocation and deallocation of disk blocks for the editor. It is important to mention that the performance of the editor should further increase with this configuration, since the present version of the data structure is built on top of a conventional file-management system. In an editing server there will be no such underlying structure.

2. The version of the editor with the new data structure can easily be implemented on other systems on which the old CHEF editor has been implemented. This requires little more than setting certain manifest constants (following CHEF's 'menu' selection methodology) to appropriate values, as described in chapter 6.

It is believed that the data structure presented here is an advanced, efficient, and new design for the workfiles of all modern multiuser line-oriented editors.

REFERENCES

- Benjamin, A.J. (1972) An Extensible Editor for a Small Machine with Disk Storage. Communications of the ACM, 15(8): 742-747.
- Bourne, S.R. (1971) A Design for a Text Editor Software - Practice and Experience, 1: 73-81.
- Coulouris G.F. and others (1976) The Design and Implementation of an Interactive Document Editor. Software - Practice and Experience, 6: 271,279.
- Deutsch, P.L. and Lampson, B.W. (1967) An Online Editor. Communications of the ACM, 10(12): 793-800.
- ECLIPSE Line Real Time Disk Operating System. Rev. 01, Data General Corporation, Sept. 1975.
- Fajman, R. and Borgelt, J. (1973) WYLBUR: An Interactive Text Editing and Remote Job Entry System. Communications of the ACM, 16(5): 314-322.
- Fraser, C.W. (1979) A Compact, Portable CRT based Text Editor. Software - Practice and Experience, 9: 121-125.
- Fraser, C.W. (1980) A Generalized Text Editor. Communications of the ACM, 23(3): 154-158.
- Hazel, P. (1980) Development of the ZED Text Editor. Software - Practice and Experience, 10: 57-76.
- Hazel, P. (1974) A General-Purpose Text Editor for OS/360. Software - Practice and experience, 4: 389-399.
- Irons, E.T. and Djorup, F.M. (1972) A CRT Editing System. Communications of the ACM, 15(1): 16-20.
- Macleod, I.A. (1977) Design and Implementation of a Display Oriented Text Editor. Software - Practice and Experience, 7: 771-778.
- Maclean, M.A. and Peck, J.E.L (1981) CHEF: a Versatile Portable Text Editor. Software - Practice and Experience, 11: 467-477.
- Maclean, M.A. and Peck, J.E.L. (1982) The CHEF Editor (user manual). Computer Science Department, University of Canterbury. 21p.
- Maclean, M.A. (1981) The WorkSpace of the CHEF Editor. 7p.

- Maclean, M.A. (1982) A Linked Data Structure for the CHEF Workspace. 5p.
- Peck, J.E.L. and Maclean, M.A. (1981) The Construction of a Portable Editor. Software - Practice and Experience, 11: 479-489.
- Pramanik, S. and Irons, E.T. (1979) A Data-Handling Mechanism of Online Text Editing System with Efficient Secondary Access. National Computer Conference. 273-277.
- Rice, D.E. and Van Dam, A. (1972) An Introduction to Information Structures and Paging Considerations for Online Text Editing Systems. Advances in Information Systems Science, 4: 93-159.
- Scheinder, B.R. and Watts, R.M. (1977) SITAR: An Interactive Text Processing System for Small Computers. Communications of the ACM, 20(6): 495-761.
- Verhofstad, J.M. (1978) Recovery Techniques for Database Systems. Computing Surveys, 10(2): 167-195.
- Van Dam, A. and Rice, D.E. (1971) Online Text Editing: A Survey. Computing Surveys, 3(3): 93-114.

APPENDIX

Listing of the new EF6 section of the CHEF editor written in the BCPL language.

SECTION. "E6" // Last modified 83-08-15

/* This section contains those procedures which are concerned with the manipulation of the workspace. The lines are stored sequentially in the text blocks which are stored randomly on disk (or file) blocks. The pointers to these lines consisting of the block numbers and the offset of the lines within the block are stored in the pointer blocks. The pointer blocks are also stored on random disk (or file) blocks. The main procedures are in this section are concerned with : fetching a line, storing a line, insertion, deletion, backup procedures and garbage collection. This section is written in a manner such that other sections need have no knowledge of how lines are stored in the workspace.

GET. "E0"

MANIFEST

```
{
  written      = -1 // true when window or text written on
  grab_bsz_ml = grab_bsz - 1 // for convenience
  block_min = (block_csz*49)/100
  block_ave = (block_csz*85)/100
  block_max = (block_csz*100)/100
  block_csz_1 = block_csz - 1 //For convenience
  block_bsz_4 = block_bsz - 4
  block_bsz_6 = block_bsz - 6
  block_bsz_8 = block_bsz - 8
  reference_count = block_csz - 2 // For garbage collection.
  following_blk = block_csz - 1 // For linking text blocks
  max_text_blk = 512
  max_pointer_blk = 50
  off_max = block_bsz/grab_bsz
  controls_per_blk = block_bsz/line_bsz
  mask = 127
  n_bits_off = 7
}
```

STATIC

```
{ controls          = 0 // vector for grabs of controls
  cur_window        = 0 // block buffer for current window
  line_base         = 0 // line base for current work-space
  max_grab          = 0 // grab beyond last used
  old_window        = 0 // block buffer for old window
  input_buffer       = 0 // block buffer for input text
  output_buffer      = 0 // block buffer for output text
  blk_n_in_buf      = -1
  need_to_restore    = TRUE
  need_old_window    = FALSE
  last_index        = 0
  cur_index          = 0
  old_index          = 0
  phys_cur_line      = 1
  cur_offset         = 0
  out_buf_off        = 0
  address            = 0
  count              = 0
  map_table          = 0
  count_old_window   = -1
```

```

backup_header      = 0
backup_window      = 0
backup_fb          = -1
base_disk_blk_n    = 6
disk_blk_n         = -1
base_out_buf_off   = 0
stack_empty        = TRUE
base_next_blk_n    = 7
next_blk_n         = 1
last_blk_n         = 0
stack_top          = -1
stack              = 0
garbage_pointers   = 0
trace_file_stream  = 0
}

```

```

LET start_ef6(n) BE
{1//trace("start_ef6:")
{ LET w_f = name_of_work_file()
  work_out_stream := find_stream(w_f, findoutput)
  UNLESS valid_stream(work_out_stream) THEN err("Work file")
  work_in_stream :=
    (sys_aos LOGOR sys_rdos LOGOR sys_rdos1) -> work_out_stream,
    find_stream(w_f, findinput)
  no_out_trim_or_control(work_out_stream)
  no_trim(work_in_stream)
  trace_file_stream := findoutput("TRACEFILE")
  input_buffer := work_space + 1
  output_buffer := input_buffer + block_csz + 1
  cur_window := output_buffer + block_csz + enu_window -> 1, 0)
    (menu_window -> 1, 0)
  address := w_space
  count := address + max_pointer_blk
  IF dedicated_disk AND 16_bit_computer THEN
    map_table := insert_lines
  backup_window := backup_garbage_space
  stack := backup_window + block_csz
  max_grab, line_base := 0,0
// trace := tracing
  IF menu_window THEN
    { old_window := cur_window + block_csz + 1
      written ! cur_window, written ! old_window := FALSE, FALSE
      cur_index, old_index := 0, -1
    }
  check_system(n, computer, 6); start_ef7(n)
  IF menu_control THEN set_up_control()
}1

```

```

AND tracing(fmt, a, b, c, d, e, f, g) BE
{1 LET o = output()
  selectoutput(trace_file_stream)
  writef(fmt, a, b, c, d, e, f, g); wrch('*N')
  selectoutput(o) }1

```

```

AND init_work_space() BE
/* This is called to initialize a fresh workspace and possibly
to read in a file. */

```

```

{1//trace("init work space:")
  IF dedicated_disk AND 16 bit computer THEN
    FOR i = base_text_blk_n TO next_blk_n
      DO deallocate_phys_blk(map_table!i)
    next_blk_n := base_next_blk_n
    stack_empty, stack_top := TRUE, -1
    last_blk_n := allocate_a_blk()
    backup_header := allocate_a_blk()
    out_buf_off := base_out_buf_off
    backup_fb := backup_header
    FOR i = 0 TO max_pointer_blk - 1 DO
      count ! i, address ! i := 0, -1
    IF dedicated_disk AND 16 bit computer THEN
      FOR i = 7 TO max_text_blk DO map_table ! i := -1
    address!0 := allocate_a_blk()
    written!cur_window, written!old_window := FALSE, FALSE
    written ! input_buffer := FALSE
    garbage_pointers := 0
    output_buffer ! reference_count := 0
    cur_index, old_index, last_index := 0, -1, 0
    phys_cur_line, cur_offset := 1, 0
    blk_n_in_buf := -10
    new_line := FALSE
    altered, cur_line, last_line, l_line1 := FALSE, 0, 0, 0
    TEST menu_fra THEN
      TEST (modifier = '(') THEN file_name
      ELSE copy_string(tmp_name, file_name)
    ELSE copy_string(tmp_name, file_name)
    IF ((modifier = 'F') LOGOR (modifier = '(')) LOGAND
      no_name THEN warn(m_name)
    UNLESS no_name THEN
      { cur_line := 1
        read_file(tmp_name) }1

AND allocate_a_blk() =
  dedicated_disk -> allocate_phys_blk(), next_blk()

AND allocate_phys_blk() = VALOF
{1//trace("allocate_phys_blk()")
  { LET temp_db = 0
    TEST stack_empty THEN
// on the ECLIPSE, only a simulation of disk block allocation
  { disk_blk_n := disk_blk_n + 1
    temp_db := disk_blk_n }
  ELSE temp_db := pop_stack()
  RESULTIS temp_db }1

AND add_entry(add_index) BE
/* For adding a new pointer block to previous ones. This is
done by shifting all elements of 'address' and 'count'
one to the right. */
{1//trace("add_entry, add point :
  FOR i = last_index TO add_index + 1 BY -1 DO
    {2 address !(i+1) := address ! i
      count !(i+1) := count ! i
    }2
  last_index := last_index + 1
  address!(add_index + 1) := allocate_a_blk()

```

```
}1
```

```
AND clear_all_flags(i) BE
//Used by 'warn' to clean up the data base after an interrupt.
TEST ~ menu_x THEN RETURN ELSE
{1//trace("clear_all_flags")
  FOR i=1 TO last_line DO
    {F LET p_w = fetch_pointer(i)
      IF p_w < 0 THEN
        { cur_window!cur_offset := -p_w
          written!cur_window := TRUE
        }
      }F }1
```

```
AND delete_entry(delete_index) BE
/*This is called to delete a pointer block. This is done by
by shifting elements of 'address' and 'count' one down.
*/
{1//trace("delete_entry, delete_point :
  TEST dedicated_disk THEN deallocate_phys_blk(delete_index)
  ELSE push_stack(address!delete_index)
  last_index := last_index - 1
  FOR i = delete_index TO last_index DO
    {2 address ! i := address ! (i+1)
      count ! i := count ! (i+1)
    }2
  }1
```

```
AND deallocate_phys_blk(b) BE
/* Only used when there is a dedicated disk. It may call a
lower level procedure to deallocate the physical blocks.*/
// on the ECLIPSE, this procedure is a null procedure.
{1 LET ignores_blk = b }1
```

```
AND delete_lines(l1l, l12, lr) BE
/* To delete a number of text lines. For this, the procedure
only shifts the pointers in the block where they reside. If
they reside in more than one block these blocks are deleted.
At the end if resulting block has less than MIN number of
pointers, 'distribute_space' is called to distribute it fairly.
*/
{1//trace("delete_lines : from=N", l_line1, l_line2)
  LET offset = pointer_ad(l1l, FALSE)
  AND t_cur_index = cur_index
  AND temp = l1l-offset+count!cur_index - 1
  IF l12 >= temp THEN
```

```
// delete the rest of the block in cur_wimdown:
{2 temp := temp - l1l + 1
  count ! t_cur_index, lr -= temp, temp
  l1l := l1l + temp - 1
  TEST count ! t_cur_index = 0
  THEN cur_index := -1
  ELSE t_cur_index += 1
```

```
// while the blocks are in the range to be deleted, delete them:
{R temp := l1l + ((t_cur_index > last_index) ->
  10, count ! t_cur_index)
```

```

TEST 112 >= temp THEN
{3  111 := temp
    lr -= count ! t_cur_index
    delete_entry(t_cur_index)
    IF old_index = t_cur_index
        THEN { old_index := -1
                written ! old_window := FALSE }
    IF old_index > t_cur_index THEN old_index -= 1
}3
ELSE BREAK
}R REPEAT
offset := 0
TEST last_index < t_cur_index
THEN IF last_index > -1 THEN
    {4 load_window(last_index)
        phys_cur_line -= 1
        cur_offset := count ! cur_index - 1
    }4
ELSE {5 cur_offset := 0
        load_window(t_cur_index)
    }5
}2
// delete a portion of the block in cur_window:
{ LET offset2 = offset + lr
  temp := count ! cur_index - offset2
  copy_cells(temp, cur_window + offset2, cur_window + offset)
  count ! cur_index := count ! cur_index - lr
  written ! cur_window := TRUE
}
IF count ! cur_index < block_min THEN
If (count ! cur_index < block_min) &
  (cur_index < last_index) THEN

// call distribute_space:
{D UNLESS old_index = cur_index+1 THEN
  { old_index := cur_index+1
    swap_block(address ! old_index)
    written ! old_window := TRUE }
  count_old_window := count ! old_index
  distribute_space(TRUE) }D
last_line := last_line - 1_range
}1 //End of delete_lines

AND distribute_space(old_x_entry) BE
/*This is called to distribute or merge two pointer blocks.
The 'old_x_entry' shows whether the pointer block stored in
old_window is a member of directory or not.
At the end it looks for any other block with less than
'block_min' no. of pointers to distribute.
*/
{1//trace("distribute_space: old_x_entry =
{ LET end_flag = TRUE
  UNLESS count_old_window < 0 THEN
  {R LET c0, c1 = count ! cur_index, count_old_window
    AND temp, temp1 = 0, 0
    written ! cur_window := TRUE
    written ! old_window := TRUE

```



```

TEST (c0+c1) > block_max THEN
{ T   IF ~ old_x_entry THEN
    { T1  add_entry(cur_index)
        old_index := cur_index + 1
        count!old_index := c1
        written !old_window := TRUE
    } T1
    temp := (c0+c1) / 2
    TEST c0 > c1 THEN
    { TT  templ := c0 - temp
        count ! cur_index := temp
        count ! old_index := templ + c1
        IF cur_offset >= temp THEN { phys_cur_line -=
                                    cur_offset
                                    cur_offset := 0 }
        FOR i = c1 - 1 TO 0 BY -1 DO
            old_window ! (i+templ) := old_window ! i
        copy_cells(templ, cur_window+temp, old_index)
    } TT ELSE

    // move extra pointers from old_index to cur_index
    { TE  templ := c1 - temp
        count ! cur_index := c0 + templ
        count ! old_index := temp
        copy_cells(templ, old_index, cur_index+c0)
        copy_cells(temp, old_index+templ, old_index)
    } TE
} T ELSE

// combine the two blocks by moving all pointers in
// old_index to cur_index
{ E  copy_cells(c1, old_index, cur_index+c0)
    count ! cur_index := c0+c1
    IF old_x_entry THEN delete_entry(old_index)
    written !old_index := FALSE
    old_index := -1
} E

//Check all the other pointer blocks, if any has less than
//'block_min' pointers in it, distribute it again.
{ temp := 1
  end_flag := TRUE
  old_x_entry := TRUE
  FOR i = 0 TO last_index - 1 DO
  { D  IF count!i < block_min THEN
      { T  end_flag := FALSE
          phys_cur_line := temp; cur_offset := 0
          //prepare the two pointer blocks for distribution.
          load_index(i+1)
          load_index(i)
          count!old_index := count!old_index
          BREAK
      } T
      temp := temp+count ! i
    } D
  } R REPEATUNTIL end_flag
  need_old_index := FALSE
} I

```

```

AND do_z() BE
/* Prints data storage information of interest to the
implementer. */
TEST ~ menu_z THEN RETURN ELSE
{1 writef("file:
  writef("pattern: /
  writef("last line:
  writes("   line       pointer       mark       length*N")
  FOR i = 1_line1 TO 1_line2 DO
  {2 check_interrupt()
    fetch_line(i)
    { LET len = line
      writef("IC          IC*N", i, fetch_pointer(i),
        (cur_tag = null -> '*S', cur_tag), len) }2 }1

AND fetch_line(i) = VALOF
/* This reads the 'i'th line from the work-space into 'line'
yielding the number of characters read. It retrieves the current
mark from the end of the line and yields the length of the line
(plus one for newline). */
{1//trace("fetch_line: i=
  { LET len = 0
    TEST i > 0 THEN
    {2 fetch_record(fetch_pointer(i))
      len := 1 + line
      IF menu_t THEN cur_tag := line
    ELSE {3 fetch_store_control(-i, FALSE)
      len := line
    RESULTIS len }1

AND fetch_pointer(i) =
/*To fetch the pointer to line 'i'. The pointer contains both
'log_block_no' and 'block_offset' for the desired line.
*/
  cur_window ! (pointer_ad(i, FALSE))

AND fetch_record(pointer_word) BE          // local to EF6
/*This reads in the block 'block' which contains the desired
line. The offset is pointed by 'block_off'. */
{1//trace("pointer_word accessed=
  { LET block_off = ((pointer_word & mask) REM off_max )
    * grab_bsz
    AND block = pointer_word >> n_bits_off
    AND line_off, remainder = 0, block_bsz_4 - block_off
    AND buffer = load_text(block)
    AND len = 2 + buffer
  //trace("fetch_record: block=N", block, block_off)
  {R IF remainder > len THEN remainder := len
    copy_bytes(remainder, buffer, block_off, line, line_off)
    len := len - remainder
    UNLESS len > 0 THEN BREAK
    line_off := line_off + remainder
    block_off, remainder := 0, block_bsz_4
    block := buffer ! following_blk
  //trace("following block fetched =
    buffer := load_text(block) }R
  REPEAT

```

```
}1
```

```
AND fetch_store_control(i, store) BE
/*If 'store' is true store the 'line' in the control line 'i'.
If 'store' is false fetch into the 'line' the control line 'i'.
*/
```

```
{1//trace("fetch_store_control :
{ LET j = i-1
  AND fb = j/ controls_per_blk
  AND b_off = ( j REM controls_per_blk)*line_bsz
  { LET buffer = load_text(fb)
    TEST store THEN
      { copy_bytes(line
        written ! buffer := TRUE }
    ELSE copy_bytes(1+buffer
      line, 0)
}
```

```
}1
```

```
AND flag_the_line(i) BE
/*The line is flagged by complementing the grab. This is used
by 'flagged_lines' of EF4. */
```

```
TEST ~ menu_x THEN RETURN ELSE
{1//trace("flag_the_line: I=
{ LET p_w = fetch_pointer(i)
  cur_window!cur_offset := -p_w; written!cur_window := TRUE
}
```

```
AND insert_existing_lines(l_off) BE
/* To insert 'r_range' number of lines from another region of the
pointer place after l_line1 */
```

```
{1//trace("insert_existing_lines")
UNLESS count_old_window < 0 THEN
{ add_entry(cur_index)
  count!old_index := count_old_window
  save_block(old_window, address!old_index, TRUE) }
{ LET p_l, c_off = phys_cur_line, cur_offset
  AND keep_old_x, keep_cur_x, w = old_index, cur_index,
    old_window

  AND r_off = 0
  TEST (cur_line = 1) & (r_line1 = 1) THEN //it is 0I...
    load_window(old_index)
  ELSE r_off := pointer_ad(r_line1, FALSE)
{ LET same_blk = keep_old_x = old_index
  AND countt = count!cur_index
  AND k_r_off = r_off
  written ! cur_window := FALSE
  IF same_blk THEN
    { keep_old_x, w := old_index, old_window
      old_window := cur_window
      old_index := cur_index
      written!old_window := TRUE }
```

```
// start insertion by moving pointers from cur_window
to old_window.
```

```
FOR i = r_line1 TO r_line2 DO
{F IF l_off > block_ave THEN
  {T save_block(old_window, address!old_index, TRUE)
    IF same_blk THEN
```

```

        { old_window := w
          written ! cur_window := FALSE
          same_blk := FALSE }
      IF cur_index > old_index THEN cur_index += 1
      add_entry(old_index)
      old_index += 1
      count!old_index := 0
      l_off := 0
      written ! old_window := TRUE
    }T
  IF r_off >= countt THEN
    {T IF modifier = 'D' THEN
      {TT count!cur_index := countt-(countt-k_r_off)
        IF k_r_off=0 THEN
          {delete_entry(cur_index)
            IF r_line1 < l_line2 THEN
              keep_old_x, keep_cur_x, old_index -=
                1, 1, 1
              cur_index:=cur_index-1 }TT
            cur_index += 1
            restore_block(cur_window, address!cur_index, TRUE)
            countt := count!cur_index
            r_off, k_r_off := 0, 0 }T
          old_window!l_off := cur_window!r_off
          r_off := r_off+1; l_off := l_off+1
          count!old_index := count!old_index + 1
        }F
      IF same_blk THEN { old_window, old_index := w, keep_old_x }

/* delete the source lines from the last block if the modifier
   is 'D', and then prepare the pointer-place to be used by
   the procedure distribute_space. */
      TEST modifier = 'D' THEN
        {D TEST cur_index = old_index
          THEN {copy_cells(count!old_index-r_off,
                        old_window + r_off,
                        old_window + k_r_off)
              count ! old_index -= (r_off - k_r_off)
              cur_index := -1
              p_l += (r_off - k_r_off)
            }
          ELSE { copy_cells(count!cur_index-r_off,
                        cur_window + r_off,
                        cur_window + k_r_off)
              count!cur_index -= (r_off - k_r_off)
              written ! cur_window := TRUE }
          TEST l_line2 < r_line1
            THEN cur_line += r_range - 1
            ELSE { cur_line -= 1
                  p_l -= (same_blk -> 0, r_range) }
        }D
      ELSE { last_line := last_line + r_range
            cur_line := cur_line + r_range - 1
          }
      TEST count_old_window < 0 THEN load_window(keep_cur_x)
      ELSE { load_window(keep_old_x)
            load_window(keep_old_x-1) }
      count_old_window := count ! old_index

```

```
phys_cur_line, cur_offset := p_l, c_off
}1
```

```
AND insert_new_lines(dot_stop) BE
/* Using the lines specified by 'f_line1' and 'f_line2' as lower
and upper limits, the function inserts text after 'l_line1',
stopping at a dot stop line, if 'dot_stop' is true, otherwise at
end of file. Sets 'cur_line' to the last line inserted. Note
that if 'f_line1' specifies the last line of the file, then the
contents of 'line' must be retrieved after end of file has been
detected. */
```

```
{1//trace("insert_new_lines")
{ LET done_ = FALSE
  r_line1, r_line2 := 0, 0; reset_byte_count()
  new_line_ := TRUE
  UNTIL done_ DO
    {U check_interrupt()
      TEST got_text(FALSE, dot_stop_) THEN
        { r_line2 := r_line2 + 1
          TEST menu_fra THEN
            IF (r_line1 = 0) THEN
              TEST is_line(f_line1, r_line2)
              THEN r_line1 := r_line2
              ELSE LOOP
            ELSE r_line1 := 1 }
          ELSE
            TEST menu_fra THEN
              { UNLESS (r_line1 = 0) LOGAND (r_line2 > 0) LOGAND
                (line_n! f_line1 = file_end) THEN BREAK
                r_line1 := r_line2; done_ := TRUE }
              ELSE BREAK
            IF menu_fra THEN IF is_line(f_line2, r_line2) THEN
              done_ := TRUE
            add_byte_count(store_line(cur_line))
            cur_line := cur_line + 1
            last_line := last_line + 1
          }U
        new_line_ := FALSE
        cur_line := cur_line - 1 }1
```

```
AND load_window(index) BE
/*The window corresponding to file block number 'address!index'
is loaded if it is not already there. In every case, however,
this window is now renamed as the current window. 'cur_window'.
*/
```

```
{1//trace("load_window: index=
  UNLESS index = cur_index THEN
    {N LET w = old_window
      UNLESS index = old_index THEN
        {3 LET blk = address!index
          swap_block(blk) }3
        old_window := cur_window; cur_window := w
        old_index := cur_index; cur_index := index }N
    }1
```

```
AND load_text(blk) = VALOF
/*This reads the block no. 'block' into 'input_buffer',
if not already there. There is no need to write the
```

previous 'input_buffer' back on disk.

```

*/
{1//trace("load_text: blk=
  UNLESS blk = blk_n_in_buf THEN
    {2 UNLESS blk = last_blk_n
      THEN
        {3 IF written ! input_buffer THEN
          { save_block(input_buffer,
            ((dedicated_disk AND 16_bit_computer)->
              map_table!blk_n_in_buf,
              blk_n_in_buf), FALSE)
            written ! input_buffer := FALSE }
          restore_block(input_buffer,
            ((dedicated_disk AND 16_bit_computer)->
              map_table!blk, blk),
            FALSE)

            blk_n_in_buf := blk
            RESULTIS input_buffer }3
        RESULTIS output_buffer }2
      RESULTIS input_buffer
    }1

```

AND make_space() = VALOF

/*This will split the block containing 'cur_line' into two from cur_line. This is used to insert new lines after cur_line.

```

*/
{1//trace("make_space")
  { LET offset = 0
    TEST cur_line > last_line THEN
      { load_window(last_index)
        phys_cur_line := cur_line
        cur_offset := count!cur_index
        written ! cur_window := TRUE
        count_old_window := -1
        RESULTIS cur_offset }
    ELSE { offset := pointer_ad(cur_line, FALSE)
          need_to_restore := FALSE
          swap_block()
          count_old_window := count!cur_index - offset
          count!cur_index := offset
          copy_cells(count_old_window, cur_window+offset,
                    old_window)

          written ! cur_window := TRUE
          need_old_window := TRUE
          old_index := cur_index + 1
          RESULTIS offset }
    }1

```

AND new_pointer_blk() BE

/*This is used in pointer_ad, to assign a new pointer block, if a new line is stored and the current pointer block is full (average no.).

```

*/
{1//trace("new_pointer_blk ")
  IF count ! cur_index > block_ave THEN
    {2 LET w = old_window
      TEST need_old_window

```

```

THEN { save_block(cur_window, address!cur_index,
                                     TRUE)
      old_index := old_index + 1 }
ELSE { LET w = old_window
      need_to_restore := FALSE
      swap_block()
      old_window := cur_window
      cur_window := w
      old_index := cur_index
      written ! old_window := TRUE }
add_entry(cur_index)
cur_index := cur_index + 1
count!cur_index := 0
cur_offset := 0
}2
count ! cur_index := count ! cur_index + 1
}1

AND next_blk() = VALOF
{1//trace("next_blk()")
{ LET temp_fb = 0
  TEST stack_empty THEN
    { temp_fb := next_blk_n
      next_blk_n += 1
      IF temp_fb > 511 THEN warn(m_over) }
  ELSE temp_fb := pop_stack()
  RESULTIS temp_fb }1

AND pointer_ad(i, store_) = VALOF
/*Yield the address of a cell containing the pointer of the
'i'th line. If 'store_' is true then the window was written.
the correct block is swapped in, if not there before, by using the
blopinter ck indexes which are in memory.
*/
{1//trace("pointer_ad: i=N", i, store_)
  IF menu_new THEN i := i + line_base
  TEST menu_window THEN
    {W LET t_cur_index, temp = cur_index, 0
     TEST phys_cur_line >= i THEN
       {2 temp := phys_cur_line - cur_offset
        WHILE i < temp DO
          {D t_cur_index := t_cur_index - 1
           phys_cur_line := temp - 1
           cur_offset := count ! t_cur_index - 1
           temp := phys_cur_line - cur_offset
          }D
        }2
      ELSE UNLESS new_line THEN
        {3 temp := phys_cur_line - cur_offset +
          count ! cur_index
         WHILE i >= temp DO
           {D
            t_cur_index := t_cur_index + 1
            phys_cur_line := temp
            cur_offset := 0
            temp := phys_cur_line + count ! t_cur_index
           }D
        }3
    }

```

```

        load_window(t_cur_index)
        cur_offset := cur_offset - phys_cur_line + i
        phys_cur_line := i
        IF store THEN
            { written ! cur_window := TRUE
              IF new_line THEN new_pointer_blk() }
        RESULTIS cur_offset
    }W
ELSE RESULTIS (cur_window + i)
}1

AND set_up_control() BE // local to EF6
/* This allows the maximum possible space for each control line
and sets its value to be harmless.
*/
TEST menu_control THEN
{1//trace("set_up_control:")
  line1, line1
  FOR i = 1 TO control_lsz - 1 DO
    { linei
      store_record()
      out_buf_off := out_buf_off + line_bsz - 8
      IF (i REM 4) = 0 THEN out_buf_off -= 4 }
    copy_string(echo_string(), line)
    store_record()
    out_buf_off := out_buf_off + line_bsz - 4
    { LET b = last_blk_n
      IF dedicated_disk THEN
        { b := allocate_a_blk()
          IF 16 bit computer THEN
            map_table!last_blk_n := b }
        save_block(output_buffer, b, FALSE)
    }1 ELSE RETURN

AND store_line(i) = VALOF
/* This writes 'line' to the i-th line of the work-space,
yielding the number of characters (plus one for new line). Note
that unless the line is a control the tag is put at the end.
Ordinary lines always are added to the end of the store place.
The last block (output_buffer) is always resident in memory.
Control lines are stored in fixed records at the begining of
the store place. Some calculations is used to find the address
of the control line.
*/
{1//trace("store_line: i=
  { LET len = 1 + line
    TEST i > 0 THEN
      {2 IF menu_t THEN line
        store_pointer(i)
        store_record() }2
      ELSE fetch_store_control(-i, TRUE)
    RESULTIS len
  }1

AND store_record() BE // local to EF6
/*This puts 'line' in the end of store place. The last block
of the store place is always in 'output_buffer'. The first
free location of it (in grab) is shown.
```



```

*/
{1//trace("store_record")
  { LET len = 2 + line
    AND line_offset, remainder = 0, block_bsz_4 - out_buf_off
    {R IF remainder > len THEN remainder := len
      copy_bytes(remainder, line, line_offset, output_buffer,
                  out_buf_off)

      output_buffer ! reference_count += 1
      len := len - remainder
      UNLESS len > 0
        {U IF out_buf_off+remainder > block_bsz_8
          THEN {store_block()
                remainder := 0
                out_buf_off := (out_buf_off + remainder +
                                grab_bsz_m1)/ grab_bsz * grab_bsz
                RETURN }U
          line_offset := line_offset + remainder
          store_block()
          remainder := block_bsz_4
        }R REPEAT
    }1

AND store_block() BE //used locally only in store_record().
{1//trace("store_block")
  { LET fb = last_blk_n
    TEST dedicated_disk AND 16_bit_computer THEN
      {T fb := allocate_a_blk()
        map_table ! last_blk_n := fb
        last_blk_n := next_blk()
        map_table ! last_blk_n := last_blk_n
      }T
    ELSE last_blk_n := allocate_a_blk()
    output_buffer ! following_blk := last_blk_n
    save_block(output_buffer, fb, FALSE)
    output_buffer ! reference_count := 0
    out_buf_off := 0
  }1

AND store_pointer(i) BE
/*This concatenates the 'last_blk_n' out_buf_off' in one word
and stores it in the i'th position in pointer place.
*/
{1//trace("store_pointer: last_blk_n =N",
//      last_blk_n, out_buf_off)
  { LET off = pointer_ad(i, TRUE)
    AND pointer_word = last_blk_n
    pointer_word := (pointer_word << n_bits_off) +
                    out_buf_off / grab_bsz
    IF pointer_word < 0 THEN warn(m over)
    cur_window ! off := pointer_word }
  }1
AND swap_block(blk) BE
/* Read the 'blk' into the old_window; This includes writing the
old block, if 'written ! old_window' is TRUE. */
{1//trace("swap_block: blk="
  IF written ! old_window THEN
    {2 LET old_fb = address!old_index
      save_block(old_window, old_fb, TRUE)
    }2
  }1

```

```

        written ! old_window := FALSE }2
    TEST need_to_restore
    THEN restore_block(old_window, blk, TRUE)
    ELSE need_to_restore := TRUE
}1

AND was_flagged(i) = VALOF
/* If the line was flagged, then unflag it and yield true;
otherwise, yield false. */
{1//trace("was_flagged: i="
{ LET p_w = fetch_pointer(i)
  IF p_w < 0 THEN { cur_window!cur_offset := -p_w
                  written!cur_window := TRUE
                  RESULTIS TRUE }
  RESULTIS FALSE }1
/*

```

-----backup procedures-----

The following procedures are concerned with fetching and storing the backup pointers used for Undo command. They are called from E8.*/

```

AND fetch_backup(l1, l2, insert, off) BE
/*This procedure is concerned with fetching the backup pointers
It is called from do b in E8. Depending on the modifier of the
'B' operator, it calls either insert_backup (if modifier = 'I')
or shift_backup (if modifier = null), to fetch the pointers.
*/

```

```

{1//trace("fetch_backup:l2=N", l2, l1)
  IF l1 < 0 THEN { copy_bytes(l+backup_window,
                              backup_window, 0,
                              line, 0)
                  fetch_store_control(-l1, TRUE)
                  RETURN }
  { LET l_r = l2 - l1
    IF ~insert THEN off := pointer_ad(l_line1, FALSE) - 1
  }

```

```

{ LET shift_backup(r, off) = VALOF
/*Shifts the pointers from backup to pointer blocks,
overwriting the old ones. This procedure is used for
undoing the commands which overwrote the original pointers
by creating new ones, such as 'R' and 'T'.
*/

```

```

{2//trace("shift_backup:r=N", r, off)
{ LET countt = count!cur_index
  written!cur_window := TRUE
  FOR b w off=0 TO r DO
  {F off := off + 1
    IF off = countt THEN
    {T phys_cur_line := phys_cur_line-cur_offset+
                        count!cur_index
      cur_offset := 0
      written!cur_window := TRUE
      load_window(cur_index+1)
      off := 0
      countt := count ! cur_index
    }T
  }

```

```

        cur_window!off := backup_window!b_w_off
    }F
    RESULTIS off
}2

IF backup_fb ~= backup_header THEN
//There are more backup blocks on disk:
{T backup_window!following_blk := 0
  save_block(backup_window, backup_fb, TRUE)
  restore_block(backup_window, backup_header, TRUE)
  {R TEST insert
    THEN off := insert_backup(block_csz_l-1, off)
    ELSE off := shift_backup(block_csz_l-1, off)
    l_r := l_r - block_csz_l
    backup_fb := backup_window!following_blk
    restore_block(backup_window, backup_fb, TRUE)
    backup_fb := backup_window!following_blk
    IF backup_fb = 0 THEN BREAK
  }R REPEAT
}T
TEST insert THEN insert_backup(l_r, off)
      ELSE shift_backup(l_r, off)
backup_fb := backup_header
garbage_pointers := 0 // to disable garbage collection
}1

AND insert_backup(l_r, off) = VALOF
/*Inserts the pointers from backup into the pointer blocks.
This procedure is used for undoing the command which deleted
the pointers, such as 'D'.
*/
{1//trace("insert_backup:l_r=N", l_r, off)
  FOR b_w_off = 0 TO l_r DO
    {F cur_window!off := backup_window!b_w_off
      count!cur_index := count!cur_index + 1
      IF off >= block_ave THEN
        {T save_block(cur_window, address!cur_index, TRUE)
          phys_cur_line := phys_cur_line-cur_offset+
                        count ! cur_index
          cur_offset := 0
          add_entry(cur_index)
          cur_index := cur_index + 1
          IF old_index >= cur_index THEN old_index += 1
          count!cur_index := 0
          off := -1
        }T
      off := off + 1
    }F
  last_line := last_line + l_r + 1
  RESULTIS off
}1

AND store_backup(11, 12) BE
/*This procedure stores the abandoned pointers in the backup, for
a possible Undo command. It is called from E8 for the operators
'CDJRT'. Before storing the pointers, it calls 'collect_garbage'
to release the text blocks for the previous pointers.
*/

```

```

{1//trace("store backup:")
  IF ll < 0 //for control line store its contents in backup:
  THEN { fetch_store_control(-ll, FALSE)
        copy_bytes(l+line
        RETURN }
  collect_garbage()
{ LET off = pointer_ad(ll, FALSE)
  AND b_w_off = 0
  AND countt = count!cur_index
  garbage_pointers := 12 - ll + 1
  {R IF off >= countt THEN
    {T phys_cur_line := phys_cur_line - cur_offset + countt
      cur_offset := 0
      load_window(cur_index+1)
      off := 0
      countt := count ! cur_index
    }T
    IF b_w_off = block_csz_l THEN
    {T LET fb = allocate_a_blk()
      backup_window!following_blk := fb
      save_block(backup_window, backup_fb, TRUE)
      backup_fb := fb
      b_w_off := 0
    }T
    backup_window!b_w_off := cur_window!off
    off := off + 1
    b_w_off := b_w_off + 1
    ll := ll + 1
  }R REPEATUNTIL ll > 12
}1
/*

```

-----garbage collection-----

The following procedures are concerned with the garbage collection.*/

```

AND collect_pointers() BE
UNLESS garbage_pointers = 0 THEN
{1//trace("collect pointers="
  IF backup_fb ~= backup_header THEN
    {T backup_window ! following_blk := 0
      save_block(backup_window, backup_fb, TRUE)
      restore_block(backup_window, backup_header, TRUE)
      {R collect_text(block_csz_l)
        garbage_pointers := garbage_pointers - block_csz_l
        backup_fb := backup_window ! following_blk
        restore_block(backup_window, backup_fb, TRUE)
        backup_fb := backup_window ! following_blk
        IF backup_fb = 0 THEN BREAK
        TEST dedicated_disk
          THEN deallocate_phys_blk(backup_fb)
          ELSE push_stack(backup_fb)
        }R REPEAT
      }T
      collect_text(garbage_pointers)
      backup_fb := backup_header
    }1

```

```

AND collect_text(l_r) BE
{1//trace("collect_text: l_r="
{ LET blk, blk_off, p_w, buffer = 0, 0, 0, 0
  FOR off = 0 TO l_r - 1 DO
    {F p_w := backup_window ! off
      blk_off := ((p_w & mask) REM off_max) * grab_bsz
      blk := p_w >> 7
      buffer := load_text(blk)
      collect_garbage(buffer, blk)
      IF blk_off + buffer
        // the rest of this line is in next block
        { blk := buffer ! block_csz_l
          buffer := load_text(blk)
          collect_garbage(buffer, blk) }
    }F
  }1

AND pop_stack() = VALOF
{1 LET blk_entry = stack ! stack_top
  stack_top := stack_top - 1
  IF stack_top < 0 THEN stack_empty := TRUE
//trace("pop_stack :
  RESULTIS blk_entry }1

AND push_stack(blk_entry) BE
{1//trace("push_stack:
  IF stack_top < block_csz_l THEN
    { stack_top := stack_top + 1
      stack ! stack_top := blk_entry
      stack_empty := FALSE } }1

AND collect_garbage(buffer, blk) BE
{1//trace("collect_garbage, reference="
{ LET kount = (buffer ! reference_count) - 1
//trace("N", blk, kount)
  buffer ! reference_count := kount
  written ! buffer := TRUE
  IF kount = 0 THEN
    TEST blk=last_blk_n THEN out_buf_off := 0
    ELSE
      { push_stack(blk)
        IF dedicated_disk THEN
          TEST 16 bit computer
            THEN deallocate_phys_blk(map_table!blk)
            ELSE deallocate_phys_blk(blk)
          written ! buffer := FALSE
        }
      }1
  .

```